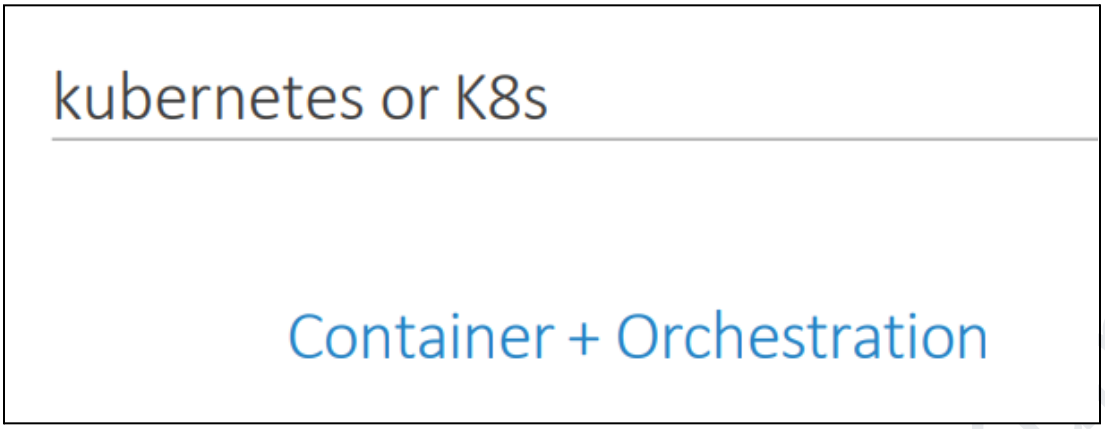
Class Notes

Date: 12/11/2023

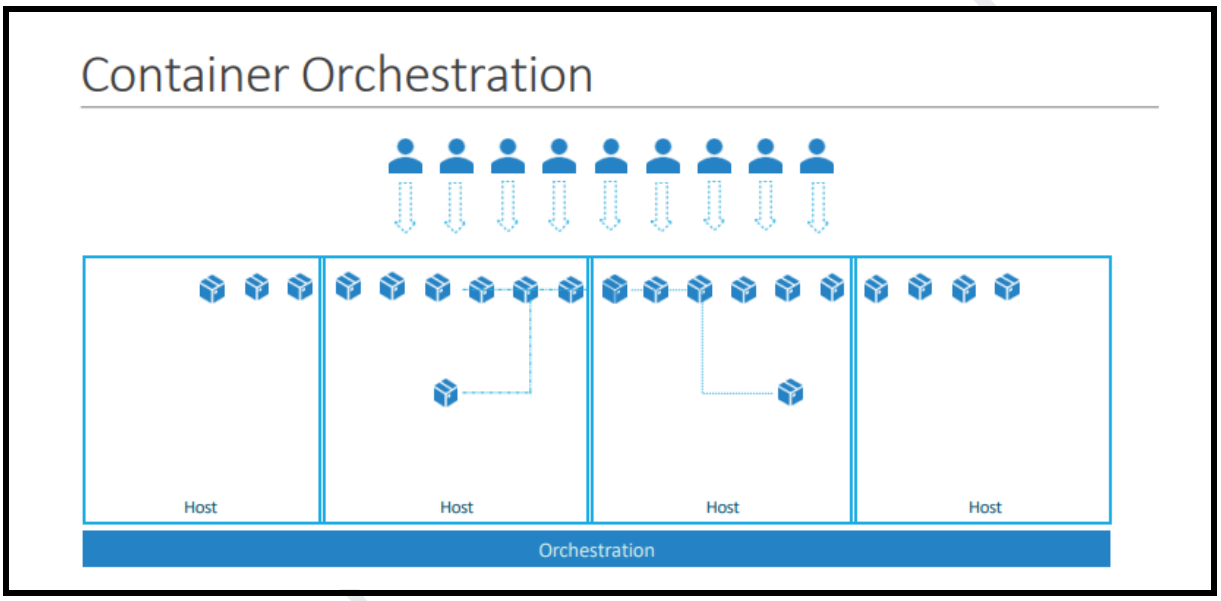
1st GRADE
KUBERNETES

Introduction: Why Kubernetes?

- Kubernetes is an open-source system for automating the deployment, scaling, and management of containerized applications.
- It was originally designed by Google and is now maintained by the Cloud Native Computing Foundation.
- Kubernetes is a go-to-platform for hosting production grade application.
- **To understand Kubernetes**, understand two things –
    - **Container and**
    - **Orchestration.**



Orchestration

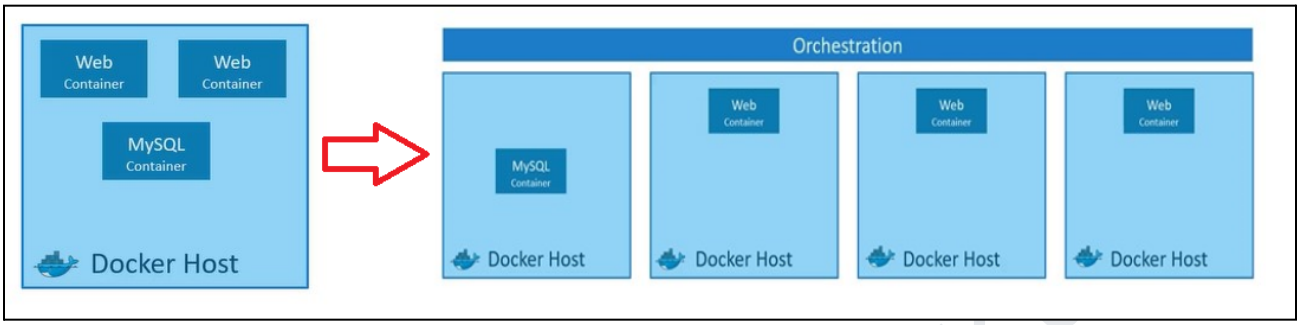


Consider now an application packaged into a docker container.

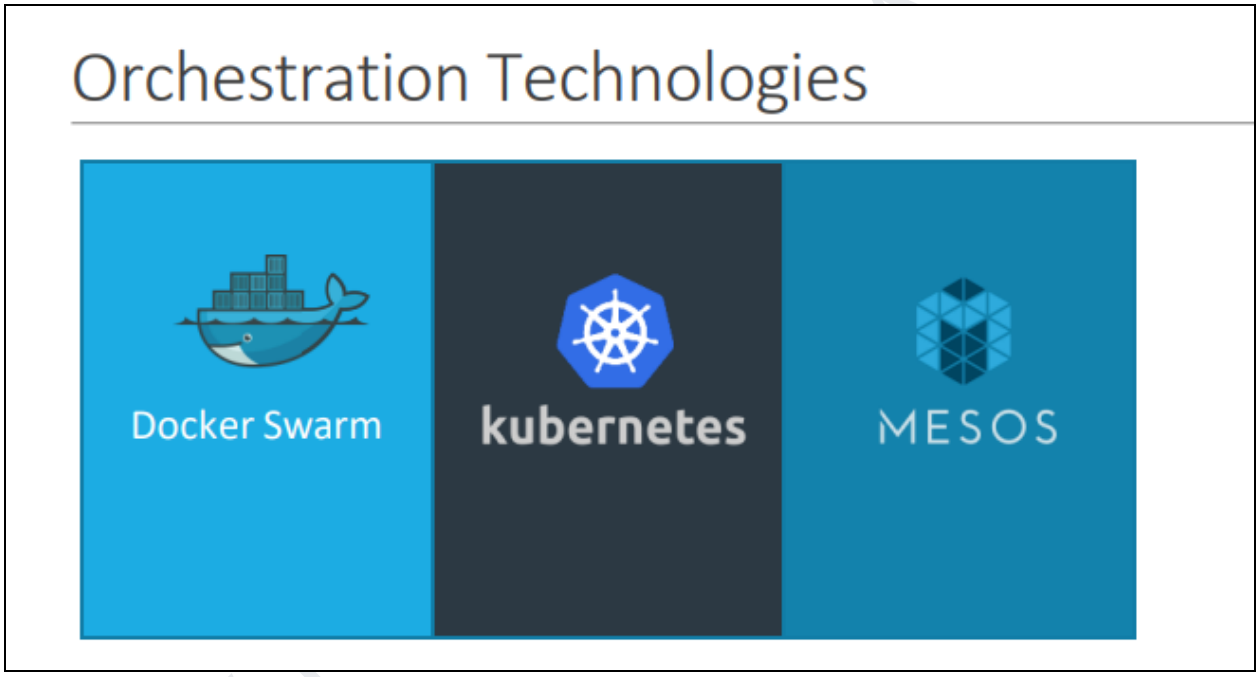
- But How do you run it in Production?
- What if the application relies on other containers such as database or messaging services or other backend services?
- What if the number of users increases and needs to scale the application?
- You would also like to scale down when the load decreases.
- To enable these functionalities we need an underlying platform with a set of resources.
- The platform needs to orchestrate the connectivity between the containers and automatically scale up or down based on the load.

Orchestration:

This whole process of automatically deploying and managing containers is known as Container Orchestration.



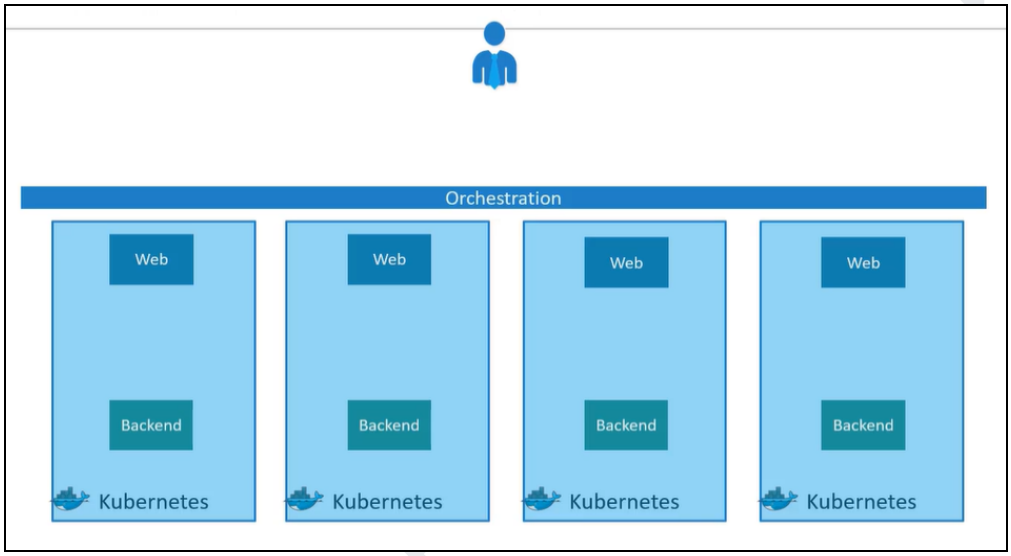
Orchestration Technologies

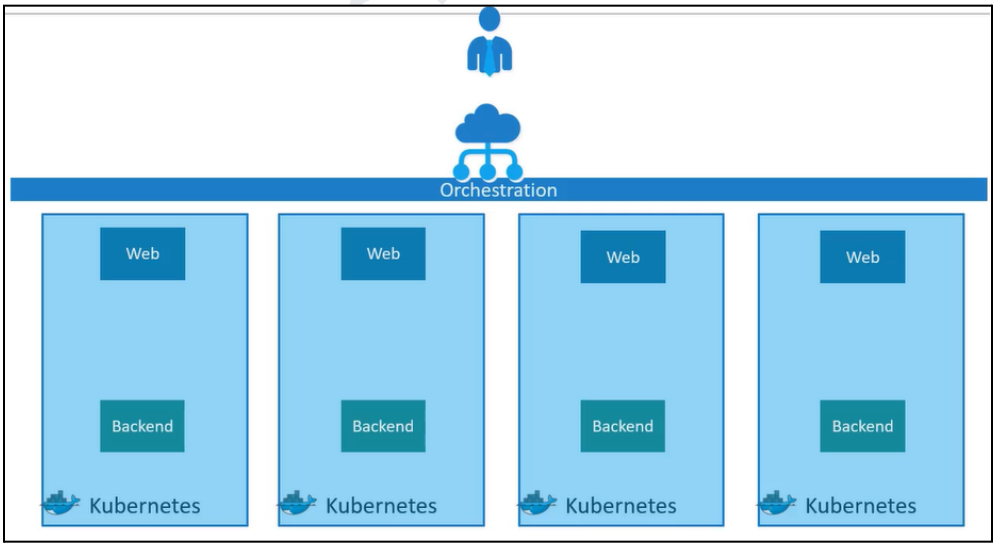


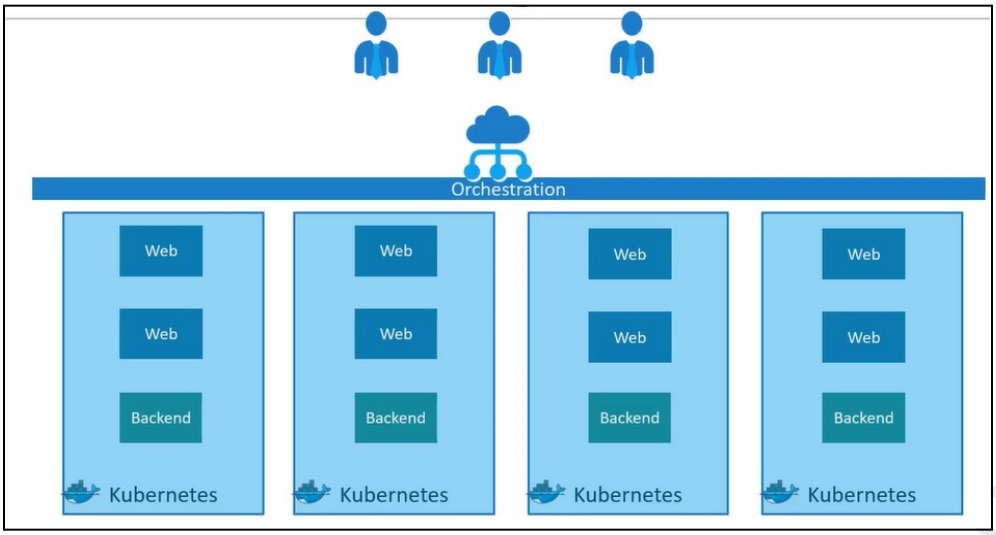
- Kubernetes is a container orchestration technology.
- There are multiple such technologies available today – Docker has its own tool called ***Docker Swarm***.
- Kubernetes from ***Google*** and ***Mesos*** from Apache.
- ***Docker Swarm*** is really easy to set up and get started, but it lacks some of the *advanced auto scaling features* required for complex applications.
- ***Mesos*** on the other hand is quite difficult to set up and get started, but supports many advanced features.

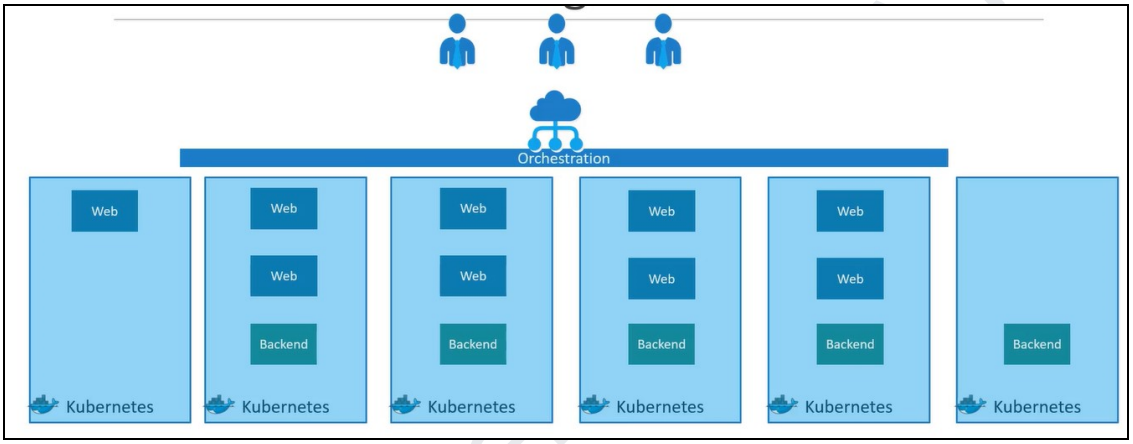
- ***Kubernetes*** - Arguably the most popular of it all – is a bit difficult to set up and get started but provides a lot of options to customize deployments and supports deployment of *complex architectures*.
- Kubernetes is now supported on all public cloud service providers like GCP, Azure and AWS and the kubernetes project is one of the top ranked projects in Github.

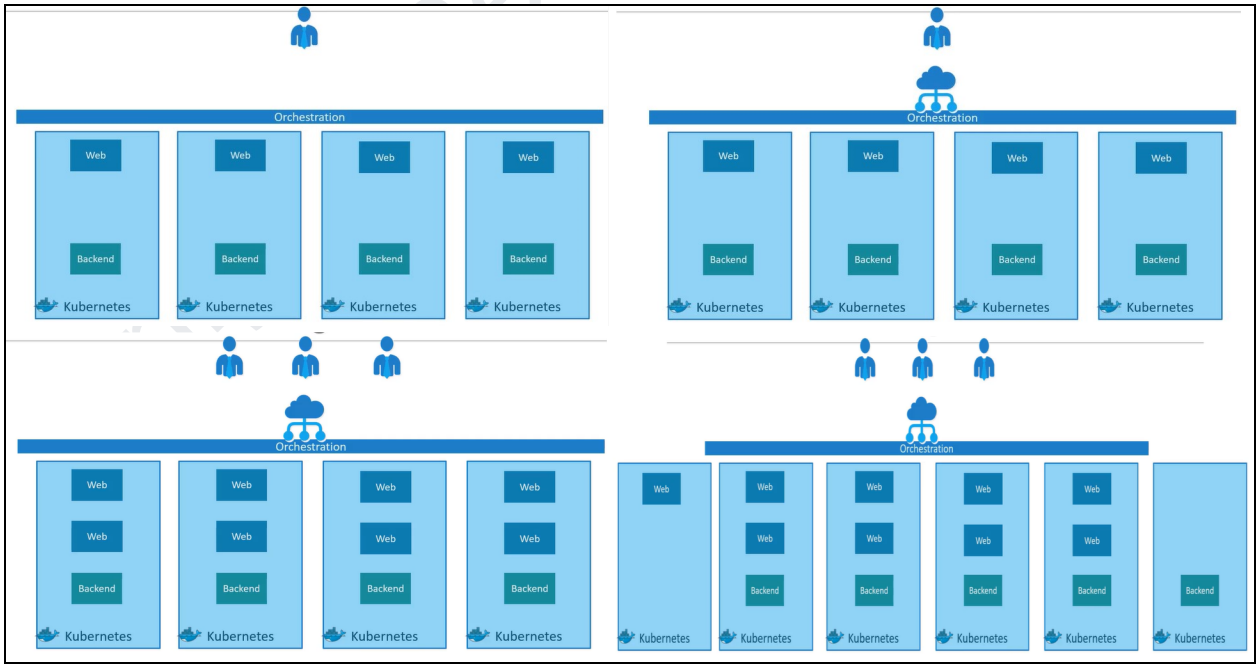
Kubernetes Advantage






Orchestration
Web
Web
Backend
Kubernetes
Kubernetes
Kubernetes
Kubernetes


Orchestration
Web
Backend
Kubernetes


Orchestration
Web
Backend
Kubernetes
Orchestration
Orchestration
Orchestration

Advantages of container orchestration

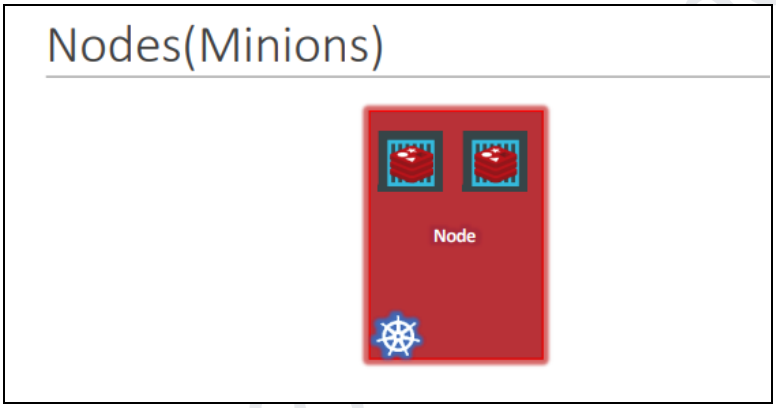
Application is now highly available as hardware failures do not bring our application down because users have multiple instances of their application running on different nodes.

- The user traffic is load balanced across the various containers.
- When demand increases, deploy more instances of the application seamlessly and within a matter of seconds and users have the ability to do that at a service level.
- When running out of hardware resources, scale the number of nodes up/down without having to take down the application.
- And do all of these easily with a set of ***Declarative Object Configuration Files***.

It is a *Container Orchestration Technology* used to orchestrate the deployment and management of 100s and 1000s of containers in a clustered environment.

Architecture

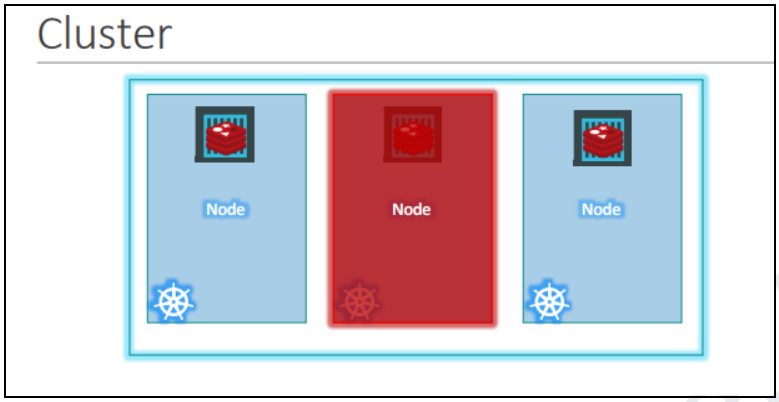
Nodes(Minions)



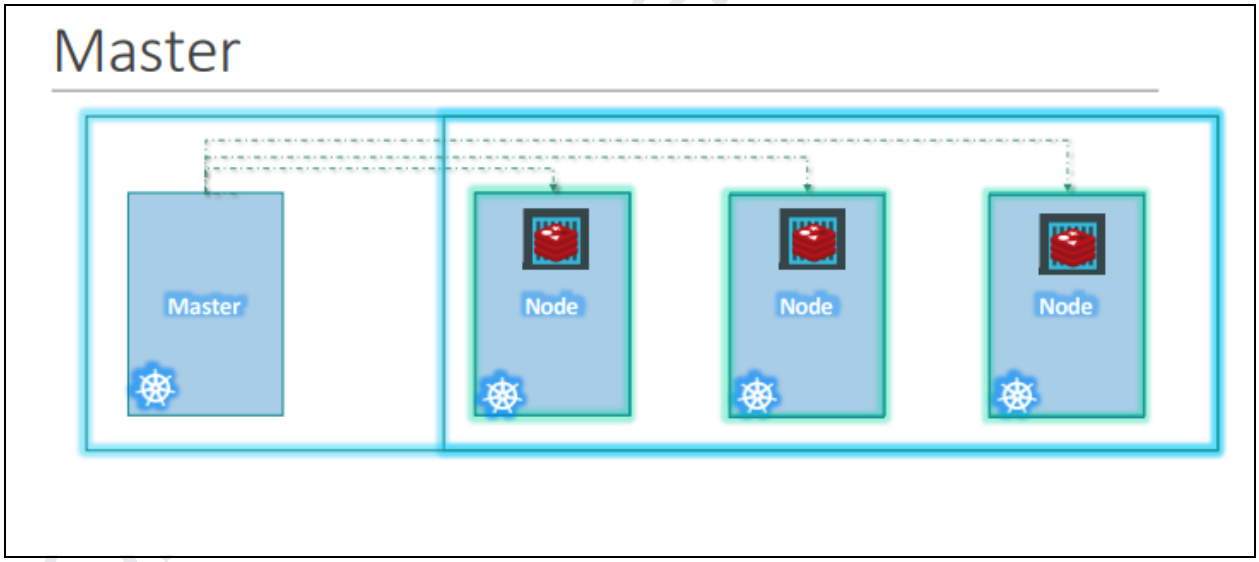
- A Node is a Machine – Physical or Virtual – on which kubernetes is installed.
- A node is a worker machine and this is where containers will be launched by kubernetes.
- It was also known as ***Minions*** in the past. (used interchangeably)
- Consider
    - If the node on which our application is running fails? Then obviously the application goes down. So need to have more than one node.

Cluster

- A cluster is a set of nodes grouped together.
- This way even if one node fails you have your application still accessible from the other nodes.
- Moreover having multiple nodes helps in sharing load as well.



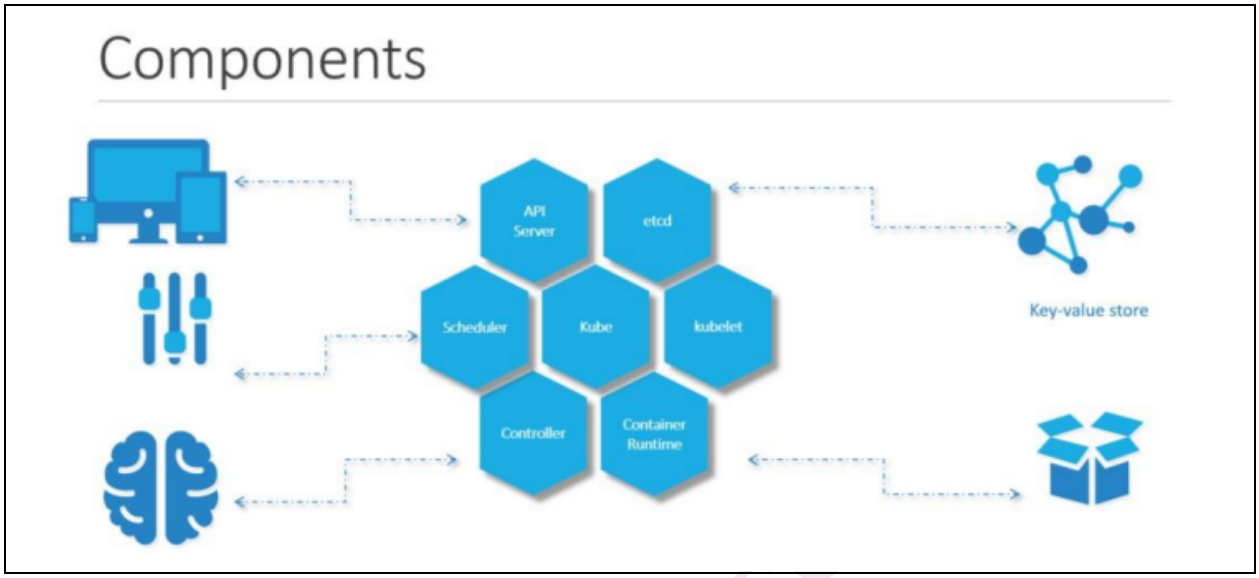
Master



- To manage a ***Cluster***, a ***Master*** is needed.
- ***Work of Master***
  - To store the information about the members of the cluster.
  - To monitor the nodes.
  - To move the workload of the failed node to another worker node.
- The master is another node with Kubernetes installed in it, and is configured as a master.

➢ The master watches over the nodes in the cluster and is responsible for the actual orchestration of containers on the worker nodes.

Components



➢ Installing Kubernetes on a System, means actually installing the following components.

- ***An API Server.***
- ***An ETCD service.***
- ***A kubelet service.***
- ***A Container Runtime, Controllers and Schedulers.***

➢ ***The API server*** acts as the front-end for kubernetes.

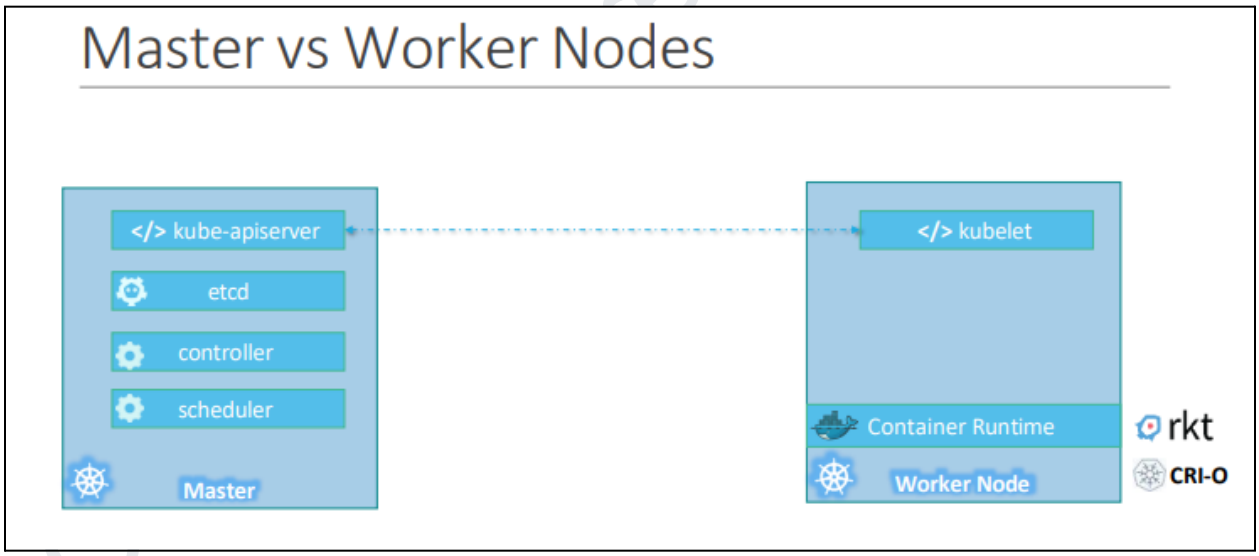
- The users, Management Devices, Command line interfaces all talk to the API server to interact with the kubernetes cluster.

➢ ***The ETCD key store***

- ETCD is a distributed, reliable key-value store used by kubernetes to store all data used to manage the cluster.
- Consider when users have multiple nodes and multiple masters in their cluster, ***etcd*** stores all that information on all the nodes in the cluster in a distributed manner.
- ETCD is responsible for implementing locks within the cluster to ensure there are no conflicts between the Masters.

- ***The scheduler*** is responsible for distributing work or containers across multiple nodes.
  - It looks for newly *created containers* and assigns them to Nodes.
- ***The controllers*** are the brain behind orchestration.
  - Responsible for ***noticing and responding*** when nodes, containers or endpoints go down.
  - The controllers make decisions ***to bring up new*** containers in such cases.
- ***The container Runtime*** is the underlying software that is used to run containers.(Here consider DOCKER).
- ***Kubelet*** is the agent that runs on each node in the cluster.
  - The agent is responsible for making sure that the containers are running on the nodes as expected.

Master vs Worker Nodes



- Seen till now,
  - Two types of servers – Master and Worker and,
  - A set of components that make up Kubernetes.

Distribution of these components across different types of servers means how does one server become a master and the other slave?

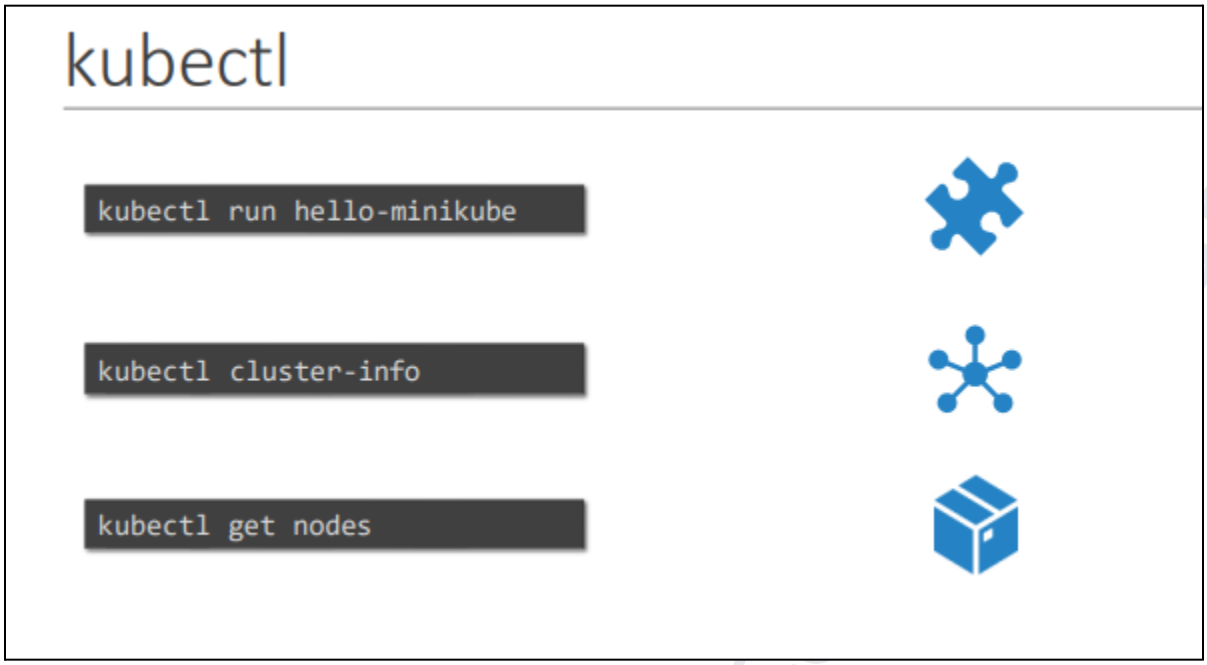
To understand what components constitute the master and worker nodes.

- The worker node (or minion) as it is also known, is where the containers are hosted.
- For example Docker containers, and to run docker containers on a system, needed a container runtime installed And that's where the container runtime falls. (Here Docker).
- *Alternatives available* (other container runtime) such as ***Rocket*** or ***CRIO***.
- The master server has the ***kube-apiserver*** and makes it a *master*.
- The worker nodes have the ***kubelet*** agent that is responsible for interacting with the master to provide health information of the worker node and carry out actions requested by the master on the worker nodes.
- *All* the information gathered is stored in a ***key-value store*** on the Master.
- The key value store is based on the popular ***etcd framework*** as discussed.
- The master also has *the controller manager and the scheduler*.

Note: There are other components as well.

- It will help to install and configure the right components on different systems when we set up infrastructure.

Kubectl



The kube Command Line Tool or kubectl or kube control

- ONE of the command line utilities known as the kube command line tool or kubectl or kube control as it is also called.
- ***The kube control tool*** is used
  - To deploy and manage applications on a kubernetes cluster,
  - To get cluster information,
  - To get the status of nodes in the cluster and many other things.

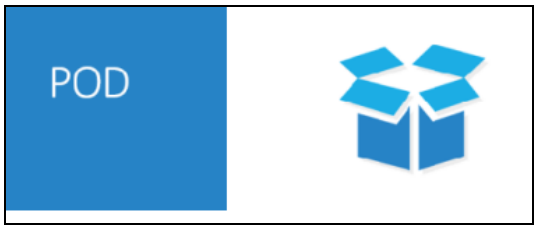
Command	***Uses***
kubectl run	To deploy an application on the cluster
kubectl cluster-info	To view information about the cluster
kubectl get pod	To list all the nodes part of the cluster
kubectl get nodes	To Show no of nodes part of the cluster
kubectl version or kubectl version --short	To Show the version of Kubernetes
kubectl get nodes -o wide	Show the flavor and version of OS where Kubernetes nodes are running

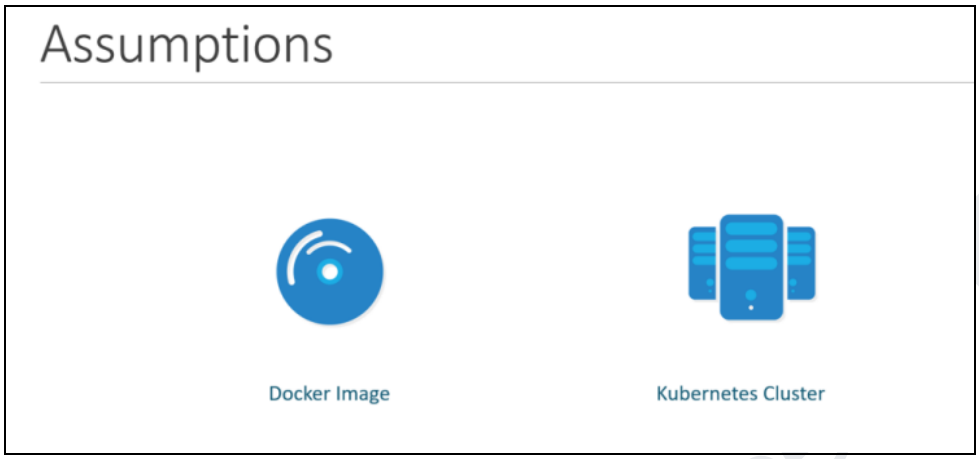
Setup

- Lots of ways to set up Kubernetes.
- Locally on our laptops or virtual machines using solutions like ***Minikube and Kubeadmin***.
- ***Minikube*** is a tool used to set up a *single instance* of Kubernetes in an All-in-one setup.
- ***Kubeadmin*** is a tool used to configure kubernetes in a *multi-node setup*.
- ***Hosted solutions*** available for setting up kubernetes in a cloud environment such as GCP and AWS.
- *checkout play-with-k8s.com* if don't have the resources or don't want to go through the hassle of setting it all up yourself, and simply want to get your hands on a kubernetes cluster instantly to play with.

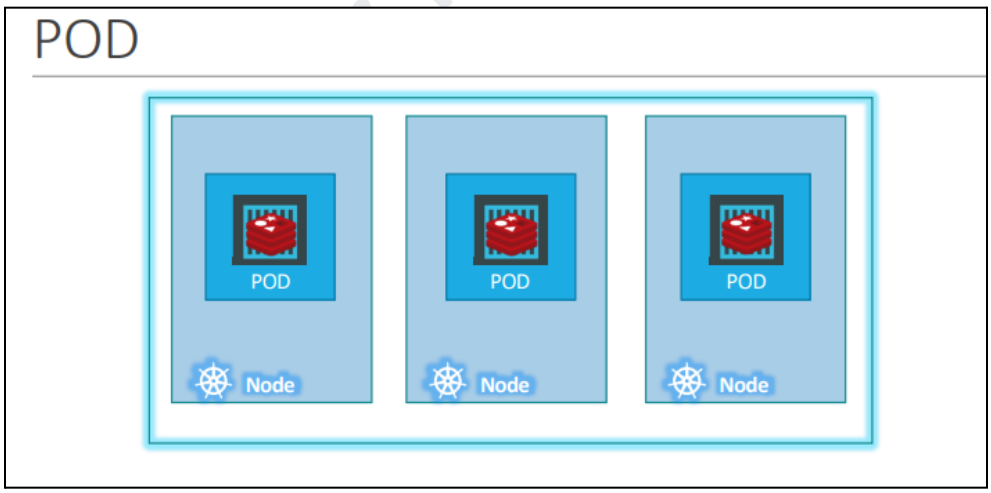
NOTE: Choose any that the best suite needs based on time and resources.

Pod

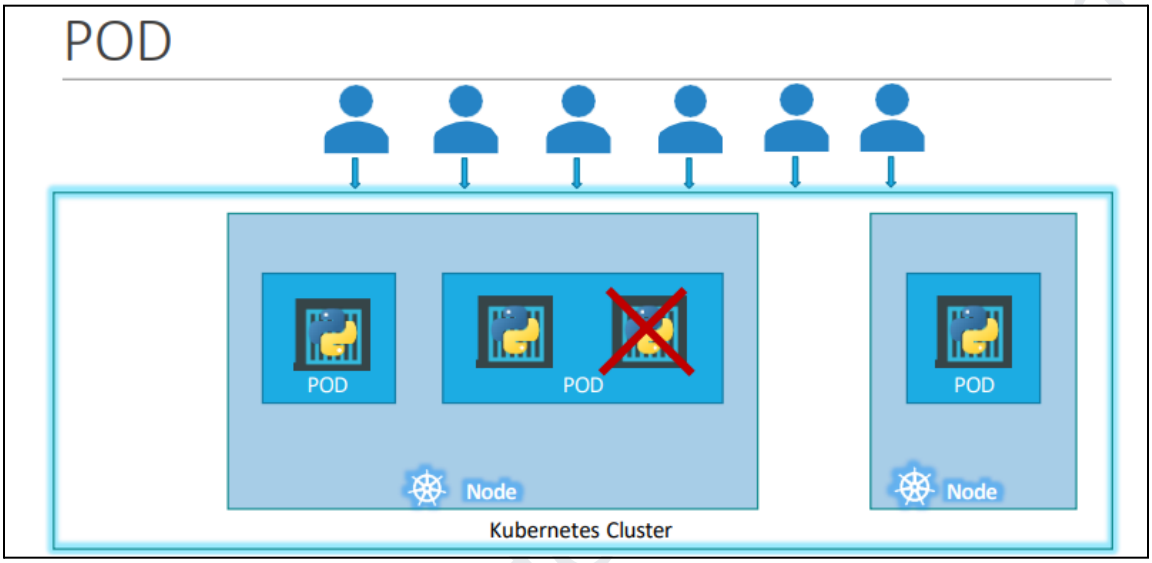




- At this point assume that the following have been setup already,
  - The application is already developed and built into Docker Images and
  - Docker Images is available on a Docker repository like Docker hub, so kubernetes can pull it down.
  - The Kubernetes cluster has already been set up and is working, may be a single-node setup or a multi-node setup.
  - All the services need to be in a running state.



- Ultimate Aim : To deploy 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 users can create in kubernetes.

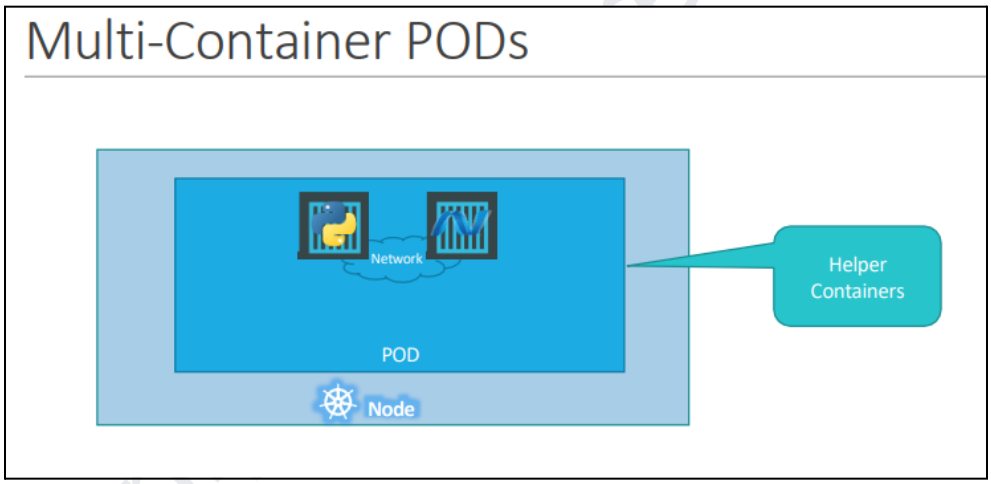


- Consider the simplest of simplest cases 👋
- Have a single node kubernetes cluster with a single instance of our application running in a single docker container encapsulated in a POD.
- What if,
  - The number of users accessing applications increases, so need to scale applications?
  - Need to add additional instances of web applications to share the load.
- Now, where would you spin up additional instances?
- Do we bring up a new container instance within the same POD?
  - No! We create a new POD altogether with a new instance of the same application.
- Here, users have *two instances of Web Application running on two separate PODs on the same kubernetes system or node*.

- ➢ Again if the user base FURTHER increases and our current node has no sufficient capacity?
    - THEN you can always deploy additional PODs on a new node in the cluster.
- ➢ You will have a new node added to the cluster to expand the cluster's physical capacity.
- ➢ SO, means that *PODs usually have a one-to-one relationship with containers* running any application.
    - To scale UP: Create new PODs and
    - To Scale down : Delete PODs.
- ➢ No need to add additional containers to an existing POD to scale our application.

Note: Think !! How to implement all of this and how we achieve load balancing between containers etc.

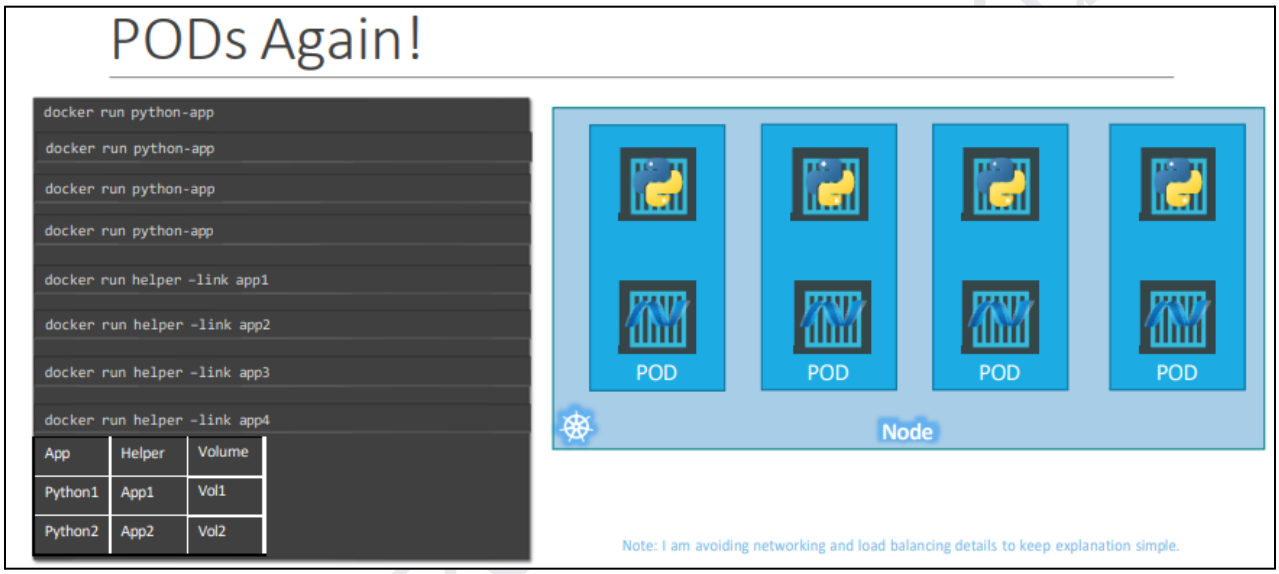
Multi Containers Pod



- ➢ As disscussed , PODs usually have a one-to-one relationship with the containers.
- ➢ But, users are not restricted to ***having a single container in a single POD***.
- ➢ A single POD CAN have multiple containers, in fact they are ***usually not*** multiple containers of the same kind.
- ➢ To scale our application, needs to create additional PODs.
- ➢ But sometimes there might be a scenario where users have a helper container that might be doing some kind of supporting task for our web application such as processing a user entered data, processing a file uploaded by the user etc.

- ➢ And users want these helper containers to live alongside their application container.
- ➢ In that case, users CAN have both of these containers part of the same POD, so that when a new application container is created, the helper is also created and when it dies the helper also dies since they are part of the same POD.
- ➢ The two containers can also communicate with each other directly by referring to each other as '***localhost***' since they share the same network namespace and the same storage space as well.

To understand PODs from a different angle.

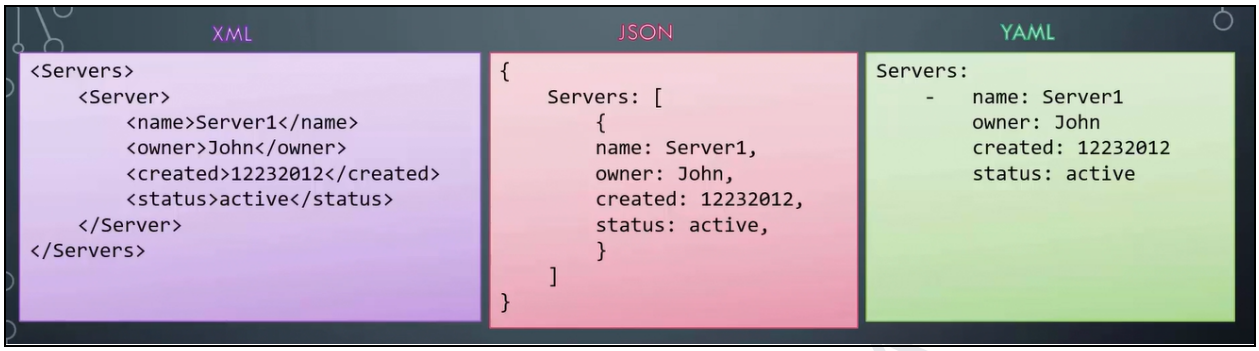


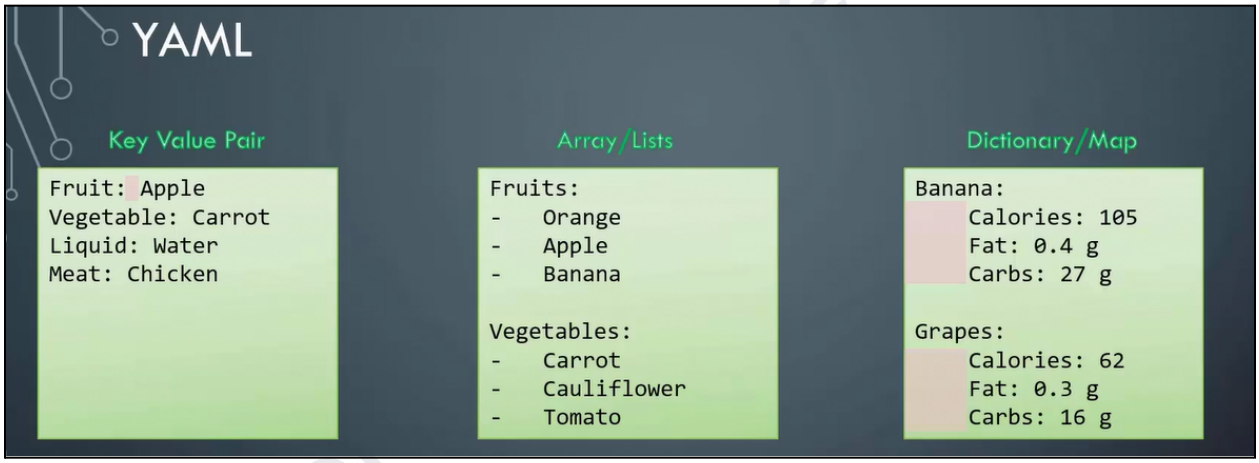
- Consider simple docker containers and no kubernetes for a moment.
- Let's assume, users were developing a process or a script to deploy their application on a docker host.
- Then they would first simply deploy their application using a simple docker run python-app command and the application runs fine and it's accessible.
- As & When the load increases they deploy more instances of their application by running the docker run commands many more times.
- This works fine and all are happy.
- Now, sometime in the future their application is further developed, undergoes architectural changes and grows and gets complex.
- Now they have new *helper containers* that help their web applications by processing or fetching data from elsewhere.

- These helper containers maintain a one-to-one relationship with their application container and thus, need to communicate with the application containers directly and access data from those containers.
- For this they need
    - To maintain a map of what app and helper containers are connected to each other.
    - To establish network connectivity between these containers ourselves using links and custom networks,
    - To create shareable volumes and share it among the containers and maintain a map of that as well.
    - And most importantly, need to monitor the state of the application container and when it dies, manually kill the helper container as well as its no longer required.
- When a new container is deployed they would need to deploy the new helper container as well.
- With PODs, kubernetes does all of this for users automatically.
- We just need to define what containers a POD consists of and the containers in a POD by default will have access to the same storage, the same network namespace, and the same fate as if they will be created together and destroyed together.
- Even if our application didn't happen to be so complex and we could live with a single container, kubernetes still requires you to create PODs.
- But this is good in the long run as our application is now equipped for architectural changes and scale in the future.
- NOTE: However, multi-pod containers are a rare use-case.

YAML:

A YAML file : Used to represent data, here Configuration Data.
Format of different data Structure like “XML”, “JSON”, & YAML.
Displayed Lists of Servers and associated information Data below.





The data in its simplest form to define such as ***Key Value Pair, an Array and a Dictionary***.

- ***Key Value Pair***
  - YAML key and value separated by a ***colon***.
  - The keys are fruit, vegetable, liquid, and meat.
  - The values are apple, carrot, water, and chicken.
  - ***Remember***, Provide a space followed by colon differentiating the key and the value.

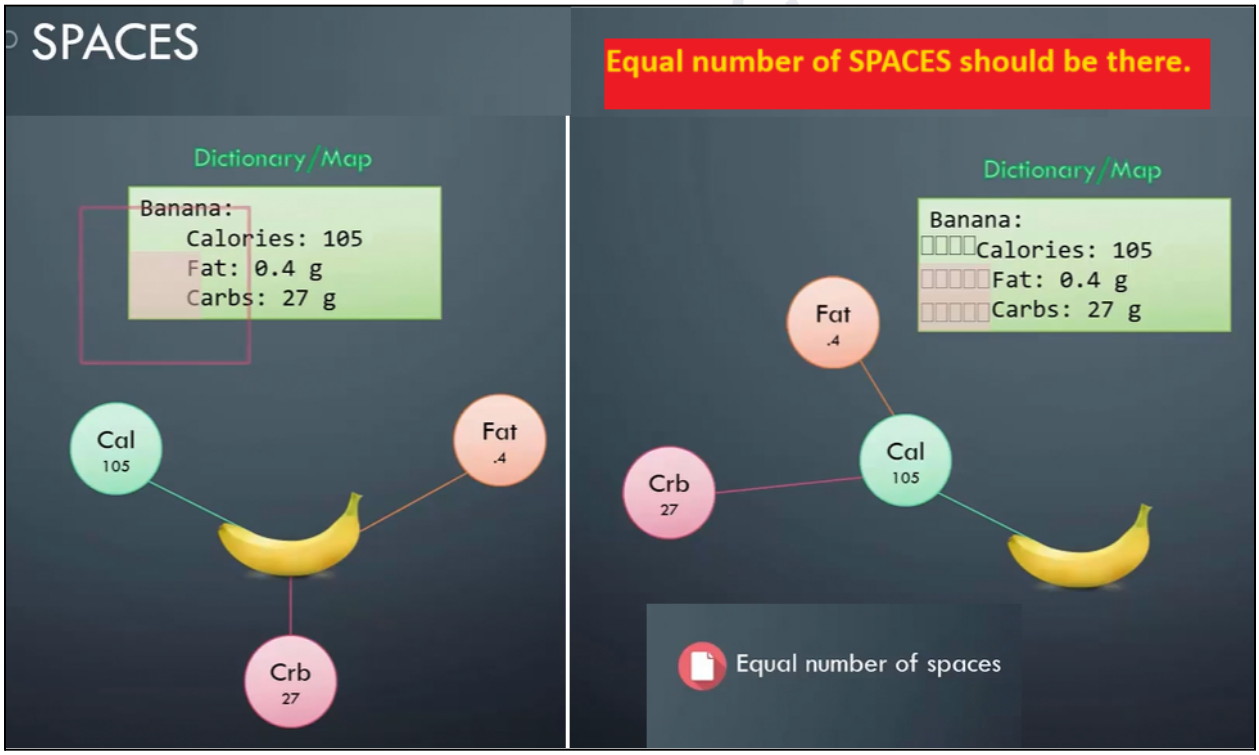
- ***An Array*** is represented as.
  - To list some fruits and vegetables.

- Define fruits followed by a ***colon***.
- On the next line enter each item with a ***dash*** in the front.
- The dash indicates that it's an ***element of an array***.

➢ ***A Dictionary***

- A dictionary is a set of properties grouped together under an item.
- Here try to represent nutrition information of two fruits
- The calories, fat, and carbs are different for each fruit.
- Notice the blank space before each item.
- Remember to provide an equal number of blank spaces before the properties of a single item so they are all aligned together.

SPACES:

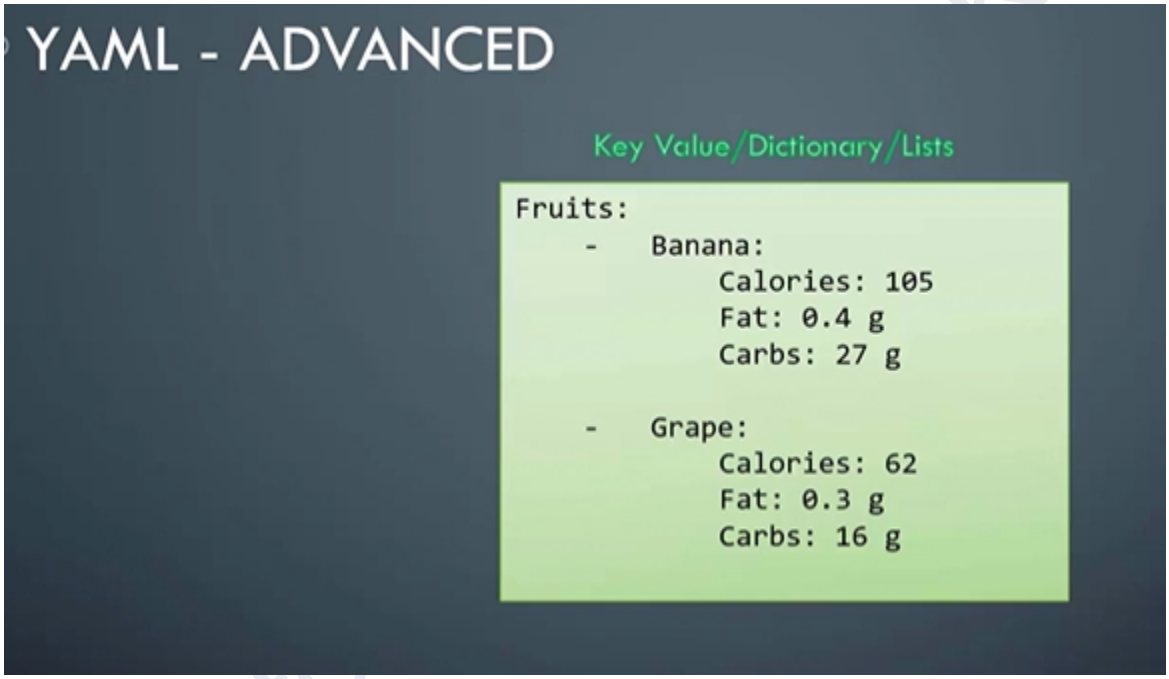


Let's take a closer look at spaces in YAML.

➔ Here, we have a dictionary representing the nutrition information of Banana.

◆ The total amount of calories, fat, and carbs are shown

- Notice the *number of spaces* before each property that indicates these key value pairs fall within the banana.
- If added extra spaces for fat and carbs.
    - Then they will fall under calories and thus become properties of calories which doesn't make any sense.
    - ***Result:*** A ***Syntax error*** which tells that mapping values are not allowed here because calories already have a value set which is 105.
- Users can either set ***a direct value or a hash map*** and cannot have both.
- The number of spaces before each property is key in YAML
- Always must ensure in the right form to represent any data correctly.



- Let's take it to another level.
    - Have lists containing dictionaries containing lists.
    - In this case, I have a ***list of fruits***.
    - The elements of the list are banana and grape.
    - Each of these elements are for *the dictionaries* containing nutrition information.

Dictionary vs list vs List of Dictionaries:

When to use a dictionary or a list.

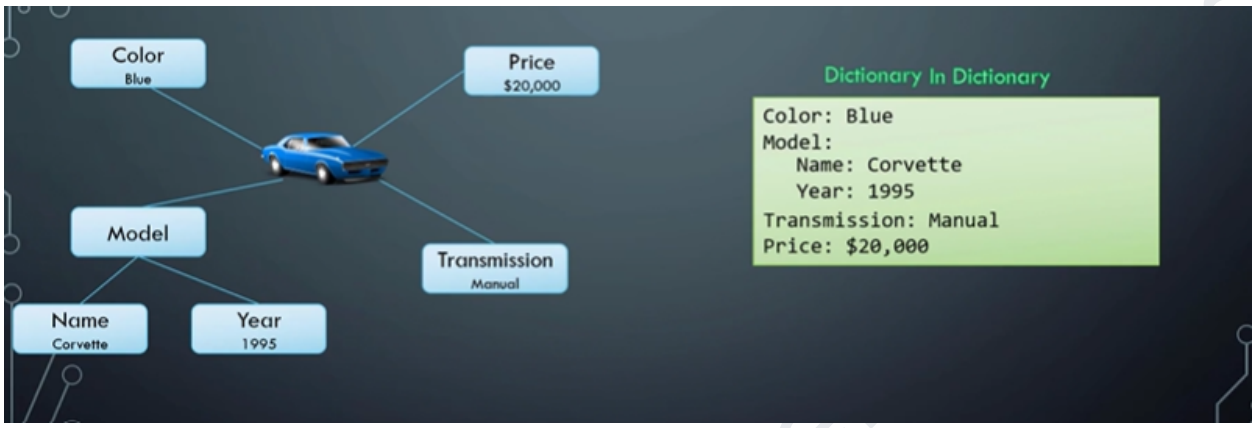
Till now understand & discussed such as XML, JSON, or YAML are used to represent data. It could be data about an organization and all of its employees and their personal details or it could be data about a school and all of its students and their grades or it could be data about an automobile manufacturing company and all of its cars and its details.
It could be anything.

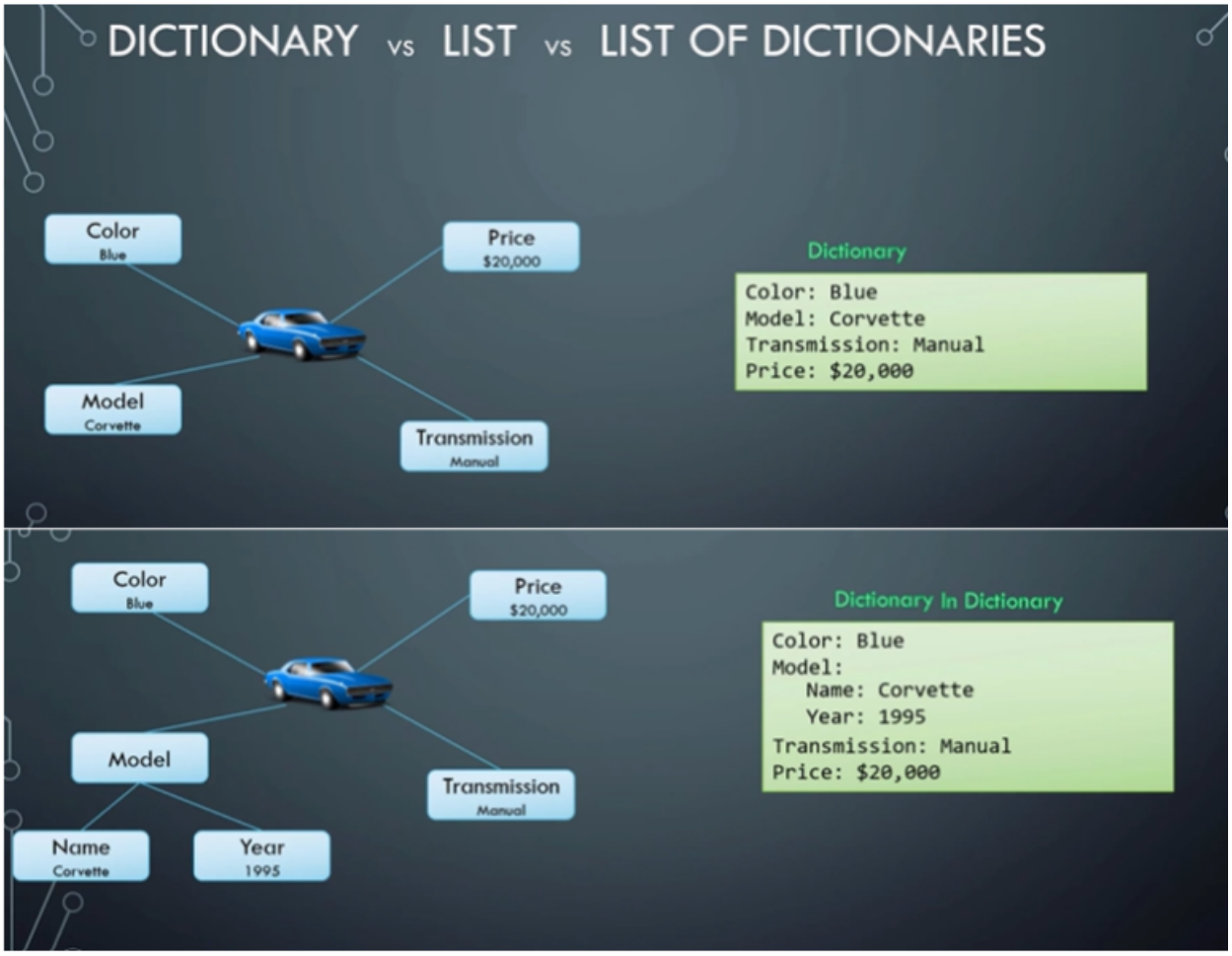
Consider an example of a car .

A car is a single object.
It has properties such as color, model, transition, and price.
To store different information or properties of a single object, we use a ***dictionary.***
In this simple dictionary properties of the car are defined in a ***key value format***.

This may not be as simple as this.

For example, in case we need to split the model further into the model name and make year, you could then represent this as a dictionary within another dictionary.
In this case, the single value of model is now replaced by a small dictionary with two properties, name and year.
This is ***a dictionary within another dictionary.***



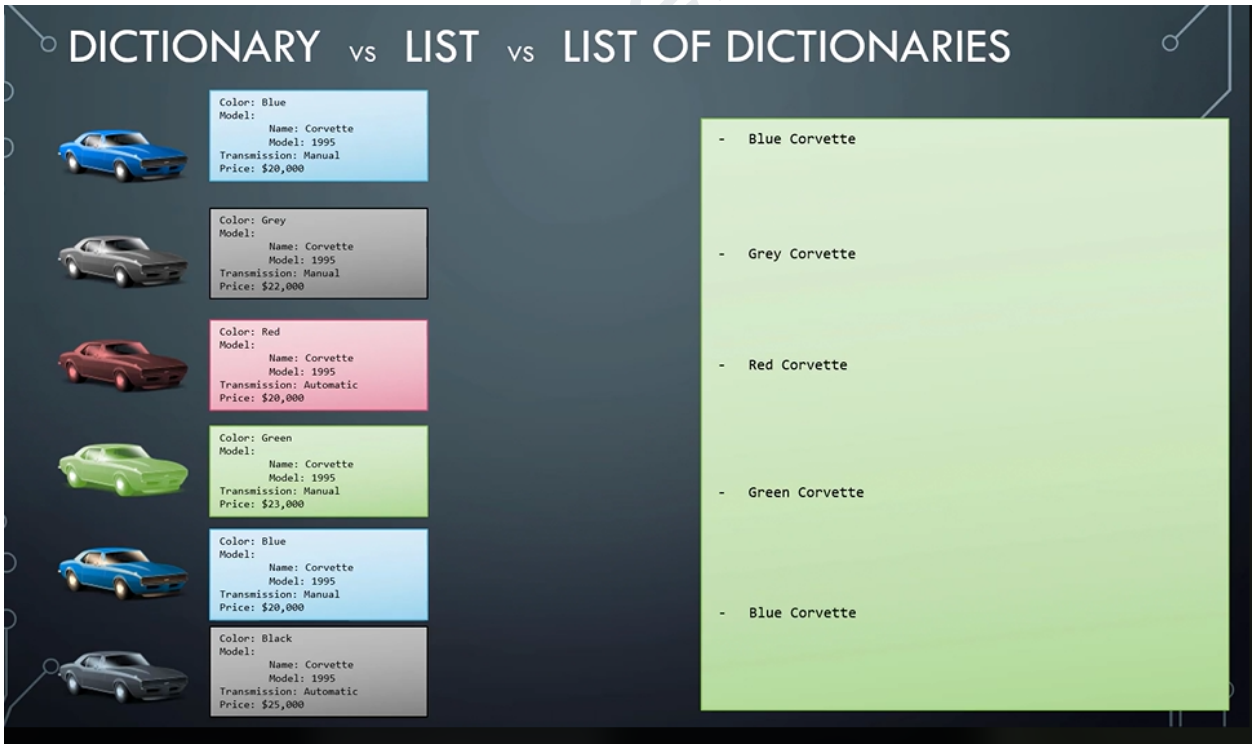


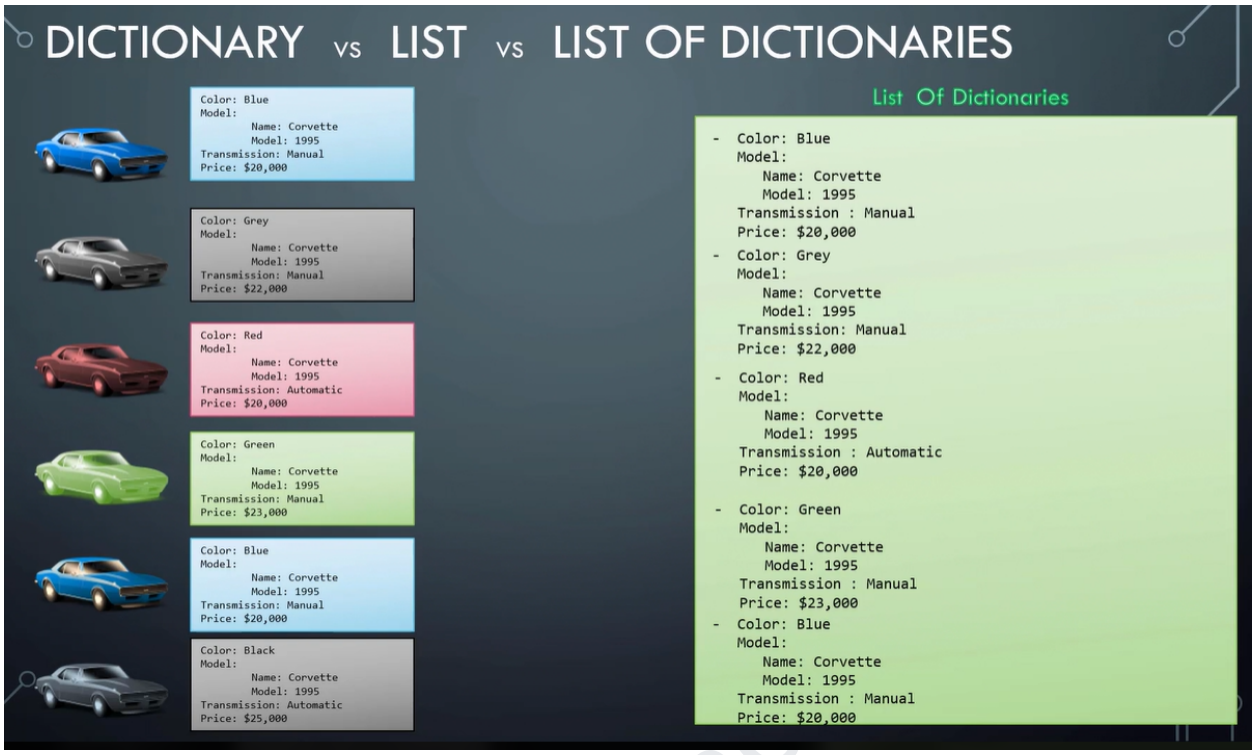
List of strings

Let's say we would like to store the name of six cars.
The names are formed by the color and the model of the car.
To store this, We would use ***a list or an array*** as it is multiple items of the same type of object.
Since we are only storing the names, we have a simple ***list of strings.***

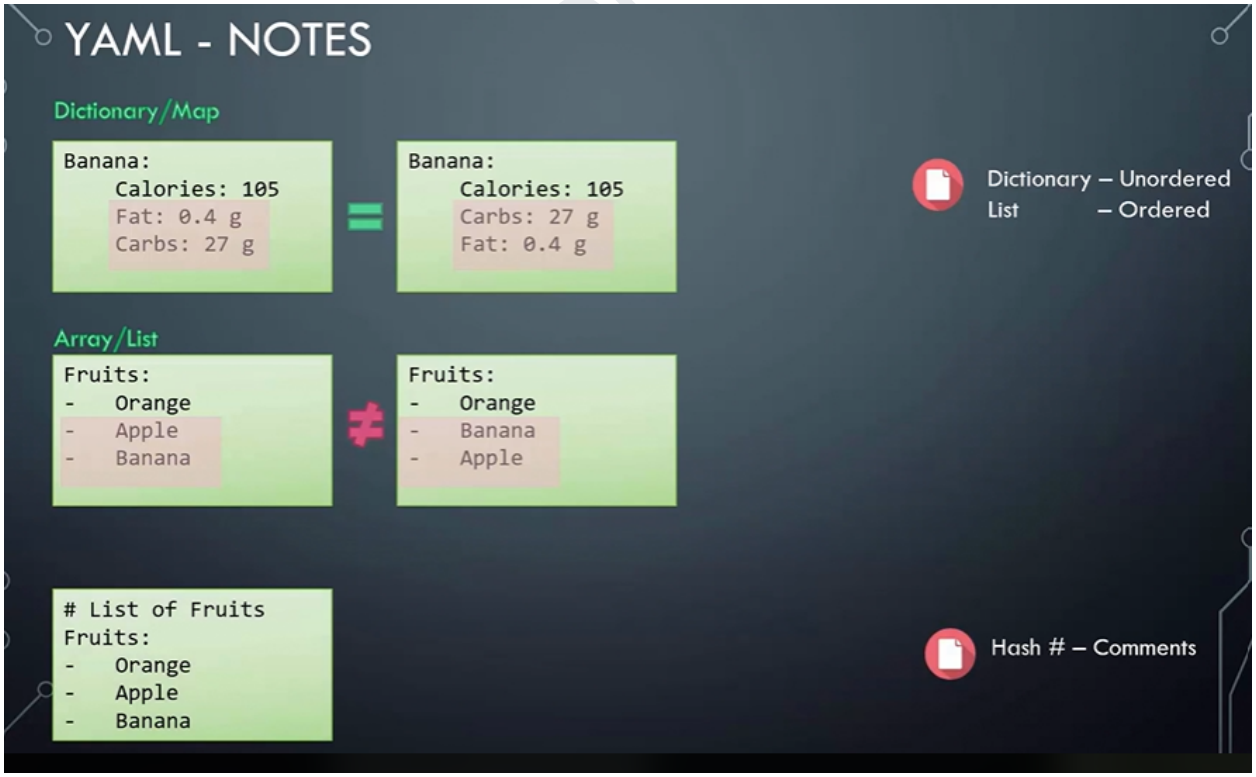
What if we would like to store all information about each car?
Everything that we listed before such as the color, model, transition, and price
We will then modify the array from a list of strings to a list of dictionaries.
We expand each item in the array and replace the name with the dictionary we built earlier.
This way, we are able to represent all information about multiple cars in a single YAML file using a list of dictionaries.
That's the difference between ***dictionary list and list of dictionaries***


DICTIONARY vs LIST vs LIST OF DICTIONARIES
Color: Blue
Model:
Name: Corvette
Model: 1995
Transmission: Manual
Price: $20,000
Color: Grey
Model:
Name: Corvette
Model: 1995
Transmission: Manual
Price: $22,000
Color: Red
Model:
Name: Corvette
Model: 1995
Transmission: Automatic
Price: $20,000
Color: Green
Model:
Name: Corvette
Model: 1995
Transmission: Manual
Price: $23,000
Color: Blue
Model:
Name: Corvette
Model: 1995
Transmission: Manual
Price: $20,000
Color: Black
Model:
Name: Corvette
Model: 1995
Transmission: Automatic
Price: $25,000
- Blue Corvette
- Grey Corvette
- Red Corvette
- Green Corvette
- Blue Corvette


DICTIONARY vs LIST vs LIST OF DICTIONARIES
Color: Blue
Model:
Name: Corvette
Model: 1995
Transmission: Manual
Price: $20,000
Color: Grey
Model:
Name: Corvette
Model: 1995
Transmission: Manual
Price: $22,000
Color: Red
Model:
Name: Corvette
Model: 1995
Transmission: Automatic
Price: $20,000
Color: Green
Model:
Name: Corvette
Model: 1995
Transmission: Manual
Price: $23,000
Color: Blue
Model:
Name: Corvette
Model: 1995
Transmission: Manual
Price: $20,000
Color: Black
Model:
Name: Corvette
Model: 1995
Transmission: Automatic
Price: $25,000
- Blue Corvette
- Grey Corvette
- Red Corvette
- Green Corvette
- Blue Corvette


DICTIONARY vs LIST vs LIST OF DICTIONARIES
Color: Blue
Model:
Name: Corvette
Model: 1995
Transmission: Manual
Price: $20,000
Color: Grey
Model:
Name: Corvette
Model: 1995
Transmission: Manual
Price: $22,000
Color: Red
Model:
Name: Corvette
Model: 1995
Transmission: Automatic
Price: $20,000
Color: Green
Model:
Name: Corvette
Model: 1995
Transmission: Manual
Price: $23,000
Color: Blue
Model:
Name: Corvette
Model: 1995
Transmission: Manual
Price: $20,000
Color: Black
Model:
Name: Corvette
Model: 1995
Transmission: Automatic
Price: $25,000
List Of Dictionaries
- Color: Blue
Model:
Name: Corvette
Model: 1995
Transmission : Manual
Price: $20,000
- Color: Grey
Model:
Name: Corvette
Model: 1995
Transmission: Manual
Price: $22,000
- Color: Red
Model:
Name: Corvette
Model: 1995
Transmission : Automatic
Price: $20,000
- Color: Green
Model:
Name: Corvette
Model: 1995
Transmission : Manual
Price: $23,000
- Color: Blue
Model:
Name: Corvette
Model: 1995
Transmission : Manual
Price: $20,000

YAML: Notes


YAML - NOTES
Dictionary/Map
Banana:
Calories: 105
Fat: 0.4 g
Carbs: 27 g
=
Banana:
Calories: 105
Carbs: 27 g
Fat: 0.4 g
Dictionary – Unordered
List – Ordered
Array/List
Fruits:
- Orange
- Apple
- Banana
≠
Fruits:
- Orange
- Banana
- Apple
List of Fruits
Fruits:
- Orange
- Apple
- Banana
Hash # – Comments

Let's take a look at some key notes Dictionary is an unordered collection whereas lists are ordered collection.
What does that mean?
The two dictionaries that you see here have the same properties for banana.
However, you can see that the order of properties, fat, and carbs do not match.
In the first dictionary, fat is defined before carbs.
In the second dictionary, carbs comes first followed by fat.
That doesn't really matter.
The properties can be defined in any order.
The two dictionaries will still be the same as long as the values of each property match.
This is not the same for lists or arrays. Arrays are ordered collection, so the order of items matter.
The two lists shown are not the same because apple and banana are at different positions.
This is something to keep in mind while working with data structures.
Also remember, any line beginning with a hash, is automatically ignored and considered as a comment.

Kubernetes Concepts – Pods, ReplicaSets, Deployments

Pods with YAML

YAML files for Kubernetes.

- Kubernetes uses YAML files as input for the Creation of objects such as *PODs, Replicas, Deployments, Services* etc.

➢ A K8s definition file contains *4 top level fields*.

1. **The apiVersion,**
2. **kind,**
3. **metadata and**
4. **spec.**

➢ These are *top level or root level* properties.

➢ Act as siblings, children of the same parent.

➢ All are REQUIRED fields, so MUST available configuration file.

➢ All of these follow a similar structure.

1. **The apiVersion,**

This is the version of the kubernetes API used to create the object.

Use the RIGHT apiVersion depends on what is trying to create.

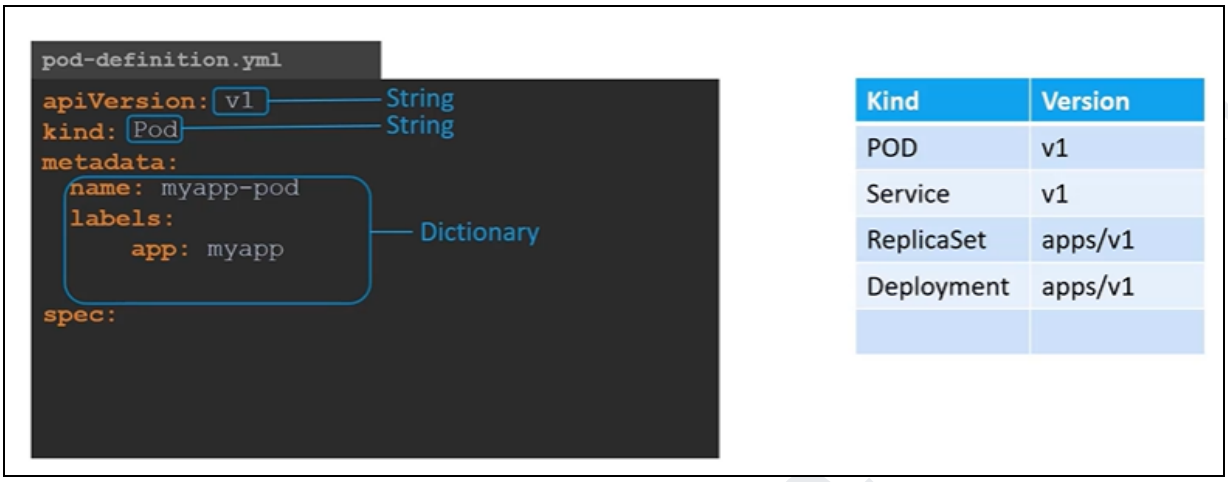
For now set the apiVersion as "***v1***" since working on PODs.

Few other possible values for this field are ***apps/v1beta1***, ***extensions/v1beta1*** etc.

2. kind

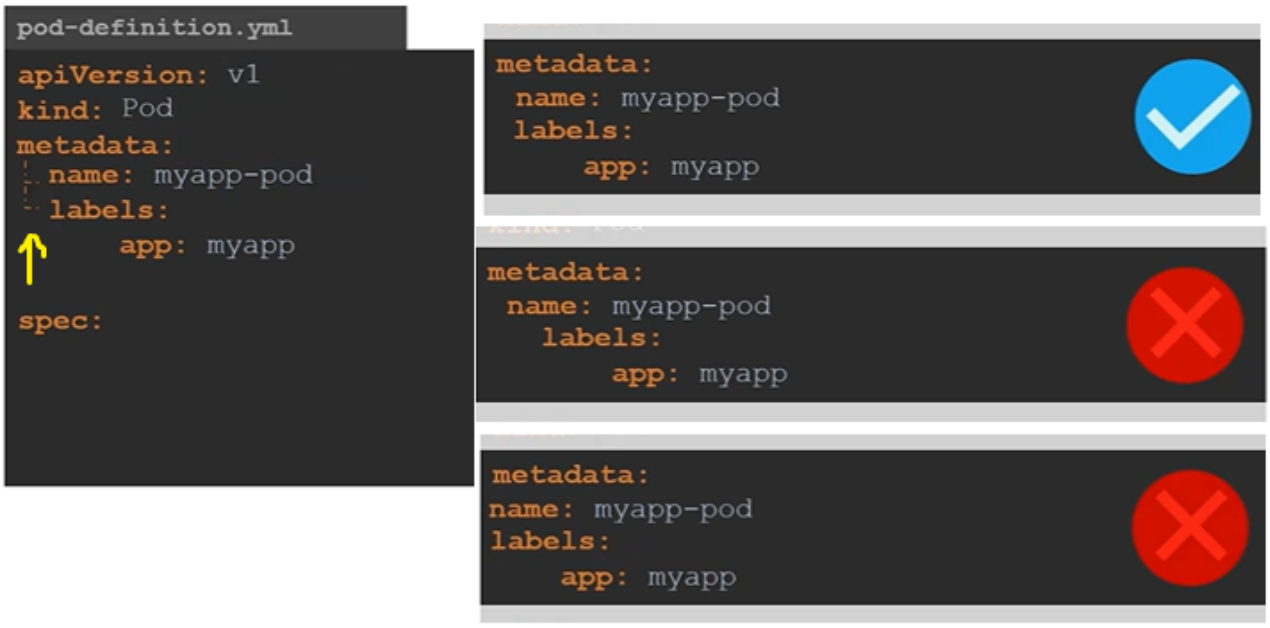
The kind refers to *the type of object trying to create*, (Here set as POD).

Some other possible values here could be ***ReplicaSet*** or ***Deployment*** or ***Service***,



Kind	Version
POD	v1
Service	v1
ReplicaSet	apps/v1
Deployment	apps/v1

3. metadata and



The next is metadata.

The metadata is data about ***the object like its name, labels*** etc.

Here the first two are specified as a string value, it is in the form of a dictionary.

So everything under metadata is intended to the right a little bit and so names and labels are children of metadata.

The number of spaces before the two properties name and labels doesn't matter, but they should be the same as they are siblings.
In this case labels has more spaces on the left than name and so it is now a child of the name property instead of a sibling.
Also the two properties must have MORE spaces than its parent, which is metadata, so that its intended to the right a little bit.
In this case all above have the same number of spaces before them and so they are all siblings, which is not correct.
Under metadata, the name is a string value – so name as your POD myapp-pod - and the label is a dictionary.
So labels is a dictionary within the metadata dictionary.
And it can have any key and value pairs as users wish.
For now I have added a label app with the value myapp.

To add other labels which helps in identifying these objects later.

Example: Let, 100s of PODs running a front-end application, and 100's of them running a backend application or a database, it will be DIFFICULT to group these PODs once they are deployed.

pod-definition.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
    type: front-end
spec:
```

- So label as front-end, back-end or database, *users* can filter the PODs based on these labels.
- IMPORTANT: Under metadata, only specify name or labels or anything that kubernetes expects as metadata.
- DON'T add any other property as per wish under this.
- However, under labels CAN have any kind of key or value pairs as seems fit.
- NOTE: Understand each parameter's expectations.

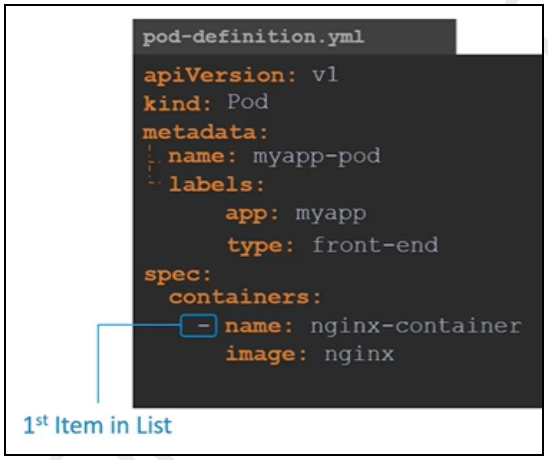
Till now only mentioned the type and name of the object, need to create, which happens to be a POD with the name myapp-pod, but haven't really specified the container or image needed in the pod.

4. **spec.**

Spec: is the specification, the last section in the configuration file.
Depending on the object they are going to create, this is where provide additional information to kubernetes pertaining to that object.
It's different for different objects, so it's important to refer to the documentation section to get the right format for each.

To create a pod with a single container.

Spec is a dictionary so add a property under it called containers, which is a list or an array.
The reason this property is a list is because the PODs can have multiple containers within them (as learned in the earlier)
Here, only planned to add a single item in the list, since planned to have only a single container in the POD.



The DASH "-" before name indicates that this is the first item in the list.
The item in the list is a dictionary, so add a name and image property.
The value for image is nginx (here) which is the name of image in Docker Repository.

Summary: Remember the 4 top level properties. apiVersion, kind, metadata and spec.
Then add values to those depending on the object needed to create.

pod-definition.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
      app: myapp
      type: front-end
spec:
  containers:
    - name: nginx-container
      image: nginx
```

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

After the file creation, run the command,

kubectl create -f <file_name>

which is pod-definition.yml and kubernetes creates the pod.

Run the kubectl Describe part command.

kubectl describe pod <pod_name>

Display information

Creation time, labels assigned, containers parts and the events associated.


```
> kubectl get pods
NAME            READY     STATUS    RESTARTS   AGE
myapp-pod       1/1       Running   0          20s
```

```
> kubectl describe pod myapp-pod
Name:         myapp-pod
Namespace:    default
Node:         minikube/192.168.99.100
Start Time:   Sat, 03 Mar 2018 14:26:14 +0800
Labels:       app=myapp
              name=myapp-pod
Annotations:  <none>
Status:       Running
IP:           172.17.0.24
Containers:
  nginx:
    Container ID:   docker://830bb56c8c42a86b4bb70e9c1488fae1bc38663e4918b6c2f5a783e7688b8c9d
    Image:          nginx
    Image ID:       docker-pullable://nginx@sha256:4771d09578c7c6a65299e110b3ee1c0a2592f5ea2618d23e4ffe7a4cab1ce5de
    Port:           <none>
    State:          Running
      Started:      Sat, 03 Mar 2018 14:26:21 +0800
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-x95w7 (ro)
Conditions:
  Type          Status
  Initialized   True
  Ready         True
  PodScheduled  True
Events:
  Type    Reason                 Age   From               Message
  ----    ------                 ----  ----               -------
  Normal  Scheduled              34s   default-scheduler  Successfully assigned myapp-pod to minikube
  Normal  SuccessfulMountVolume  33s   kubelet, minikube  MountVolume.SetUp succeeded for volume "default-token-x95w7"
  Normal  Pulling                33s   kubelet, minikube  pulling image "nginx"
  Normal  Pulled                 27s   kubelet, minikube  Successfully pulled image "nginx"
  Normal  Created                27s   kubelet, minikube  Created container
  Normal  Started                27s   kubelet, minikube  Started container
```

DEMO:

```
# vim pod.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
    tier: frontend
spec:
  containers:
  - name: nginx
    image: nginx
~
~
```

```
# cat pod.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
    tier: frontend
spec:
  containers:
  - name: nginx
    image: nginx
```

```
# kubectl apply -f pod.yaml
```

```
admin@ubuntu-server kubernetes-for-beginners #
admin@ubuntu-server kubernetes-for-beginners # kubectl apply -f pod.yaml
pod/nginx created
admin@ubuntu-server kubernetes-for-beginners #
admin@ubuntu-server kubernetes-for-beginners #
admin@ubuntu-server kubernetes-for-beginners # kubectl get pods
NAME    READY   STATUS              RESTARTS   AGE
nginx   0/1     ContainerCreating   0          7s
admin@ubuntu-server kubernetes-for-beginners #
admin@ubuntu-server kubernetes-for-beginners #
admin@ubuntu-server kubernetes-for-beginners # kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          9s
admin@ubuntu-server kubernetes-for-beginners #
```

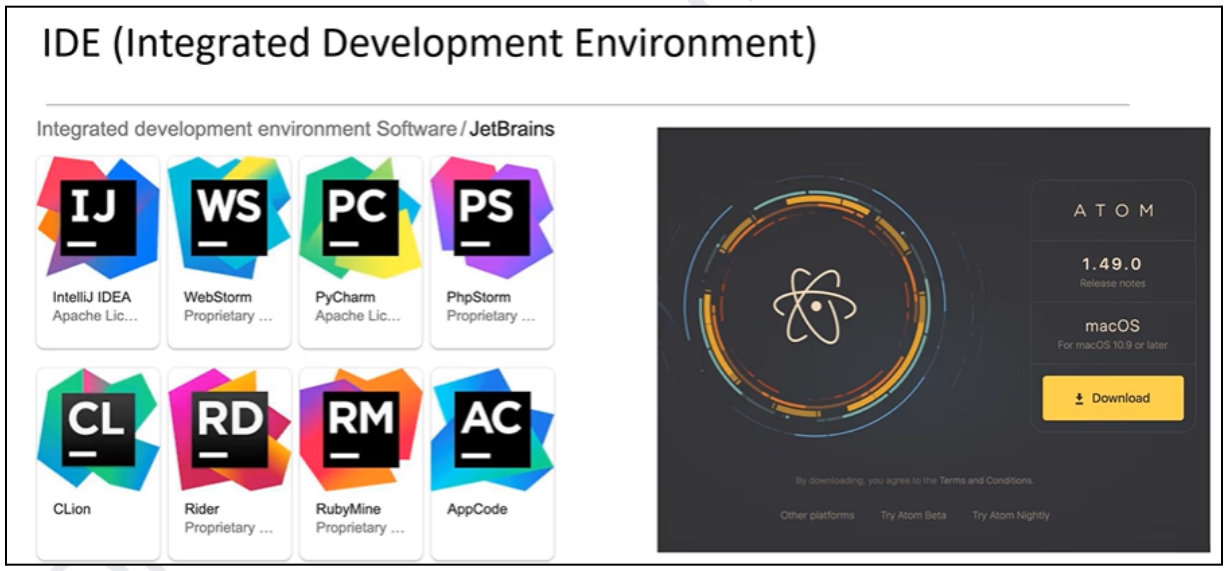
```
# kubectl describe pod nginx
```

pod.yml File:

```
apiVersion: v1
kind: Pod
metatdata:
  name: nginx
  labels:
    app: nginx
    tier: frontend
spec:
  containers:
  - name: nginx
    image: nginx
```

Tips: YAML – Tips

- Different IDE Tools for creating yml files to get help in syntax.



NOTE: Use VS IDE with YML extension enabled.

Replication Controllers and ReplicaSets

Kubernetes Controllers.

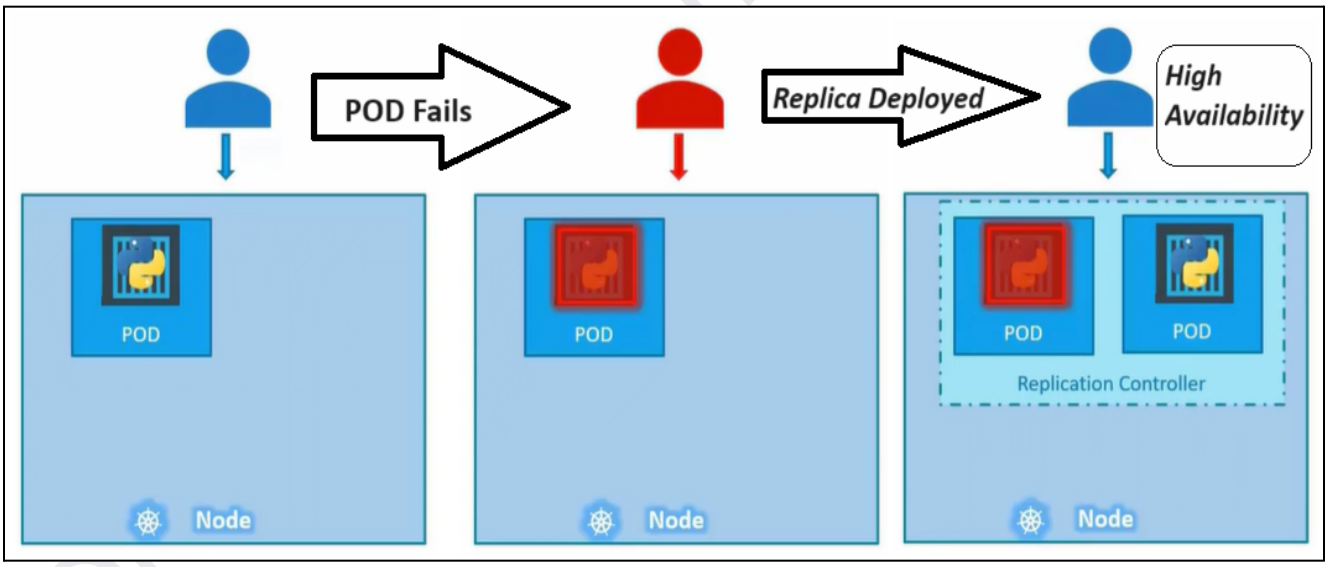
Controllers are the brain behind Kubernetes.

They are processes that monitor kubernetes objects and respond accordingly.

What is a replica and why need a replication controller?

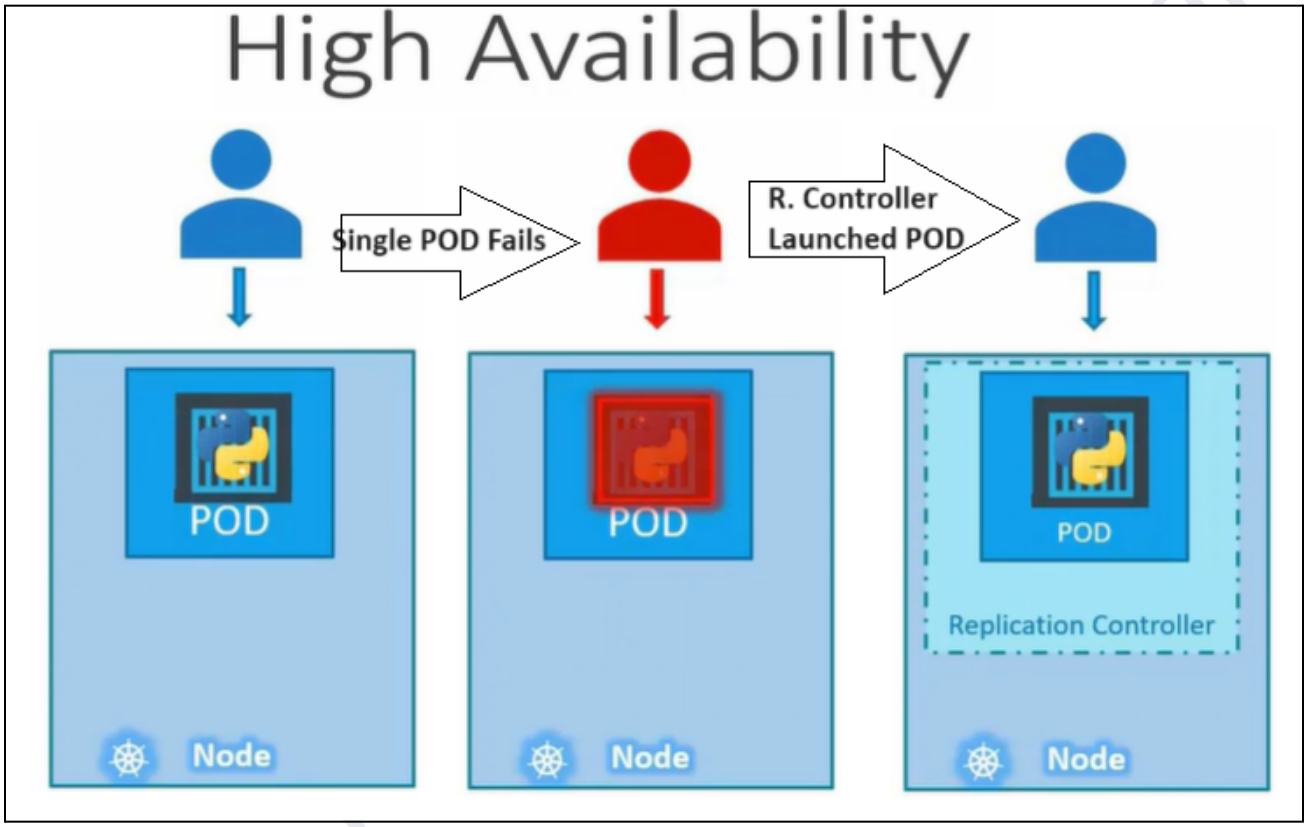
In Case of Multi POD:

- Users have a single POD running any application, & app crashes and the POD fails?
- Users will no longer be able to access applications.
- To prevent users from losing access to applications, need to have more than one instance or POD running at the same time.
- That way if one fails users still have application running on the other one.
- The replication controller helps us run multiple instances of a single POD in the kubernetes cluster thus providing ***High Availability***.

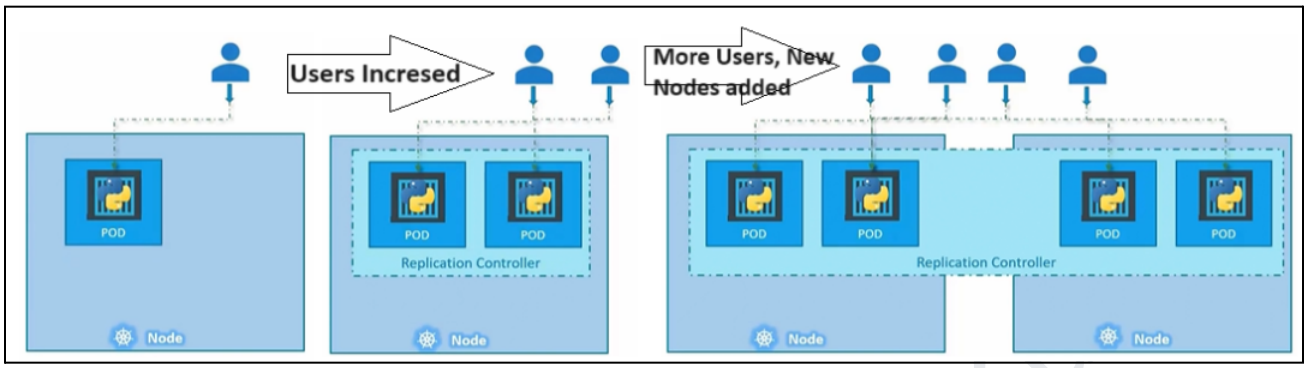


In Case of SINGLE POD:

- ➢ Note: A Replication Controller also can be used in case of a single POD & the replication controller helps by automatically bringing up a new POD when the existing one fails.
- ➢ Thus the replication controller ensures that the specified number of PODs are running at all times, even if it's just 1 or 100.



Load Balancing and Scaling



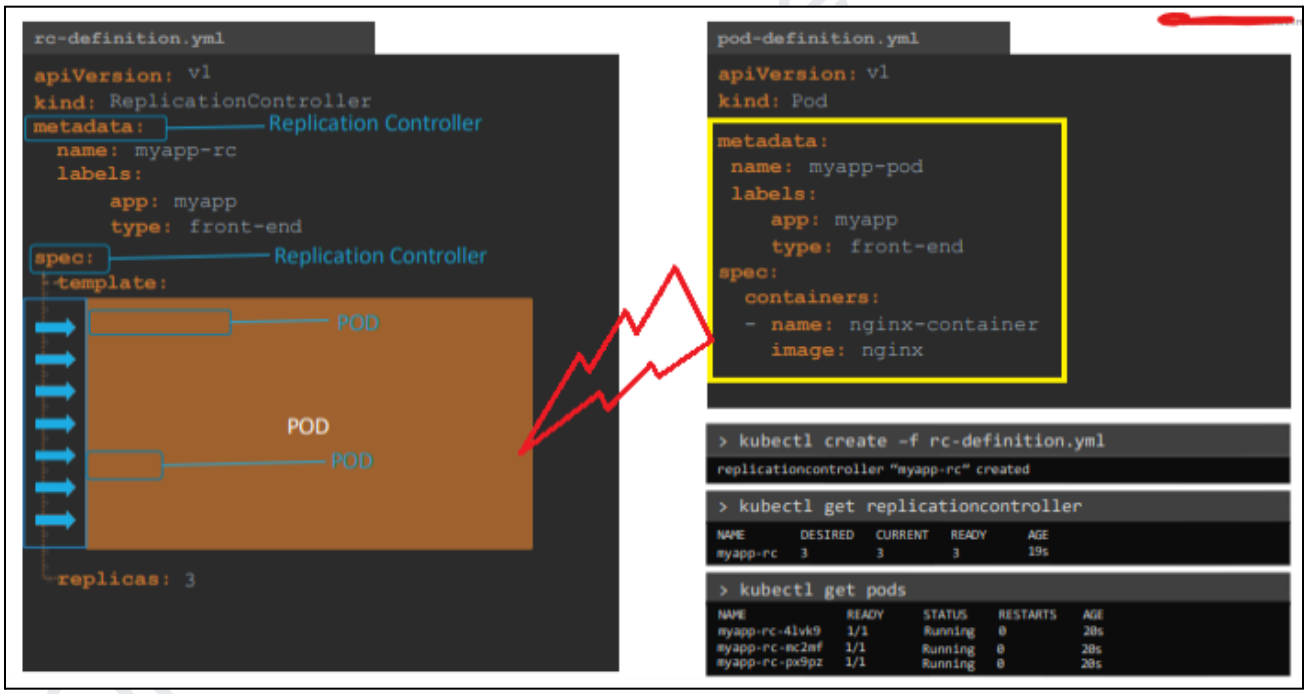
Reason 2:

- ➢ Need replication controller is to create multiple PODs 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 increases 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 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 pods on different nodes as well as scale our application when the demand increases.

Replication Controller and Replica Set.

- There are two similar terms: ***Replication Controller and Replica Set***.
- Both have the same purpose but they are not the same.
- Replication Controller is the ***older technology*** that is being replaced by ***Replica Set***.
- Replica set is the new recommended way to set up replication.
- However, whatever we discussed previously remains applicable to both these technologies.
- There are minor differences in the way each works and we will look at that in a bit.
- As such we will try to stick to Replica Sets in all of our demos and implementations going forward.

Replication Controller



How to create a replication controller?

- Renaming a replication controller definition file created earlier as *rcdefinition.yml*.
- As with any kubernetes definition file, we will have 4 sections.
- The apiVersion, kind, metadata and spec.
- The apiVersion is specific to what we are creating.
- In this case replication controller is supported in kubernetes apiVersion v1.

- ➔ So we will write it as v1.
- ➔ The kind = ReplicationController.
- ➔ Under metadata, we will add a name and we will call it myapp-rc.
- ➔ And we will also add a few labels, app and type and assign values to them.
- ➔ So far, it has been very similar to how we created a POD in the previous section.
- ➔ The next is the most crucial part of the definition file and that is the specification written as spec.
- ➔ For any kubernetes definition file, the spec section defines what's inside the object we are creating.
- ➔ In this case we know that the replication controller creates multiple instances of a POD.
- ➔ But what POD?
- ➔ We create a "***template***" section under spec to provide a POD template to be used by the replication controller to create replicas.
- ➔ Now how do we DEFINE the POD template?
- ➔ It's not that hard because, we have already done that in the previous exercise.
- ➔ Remember, we created a pod-definition file in the previous exercise.
- ➔ We could re-use the contents of the same file to populate the template section.
- ➔ Move all the contents of the pod-definition file into the template section of the replication controller, except for the first two lines – which are *apiVersion and kind*.
- ➔ Remember whatever we move must be UNDER the template section.
- ➔ Meaning, they should be intended to the right and have more spaces before them than the template line itself.
- ➔ Looking at our file, we now have two metadata sections – one is for the Replication Controller and another for the POD and we have two spec sections – one for each.
- ➔ We have nested two definition files together.
- ➔ The replication controller being the parent and the pod-definition being the child.
- ➔ Now, there is something still missing.
- ➔ We haven't mentioned how many replicas we need in the replication controller.
- ➔ For that, add another property to the spec called replicas and input the number of replicas you need under it.
- ➔ Remember that the template and replicas are direct children of the spec section.
- ➔ So they are siblings and must be on the same vertical line : having equal number of spaces before them.

- ➔ Once the file is ready, run the kubectl create command and input the file using the –f parameter.
- ➔ The replication controller Is created. When the replication controller is created it first creates the PODs using the pod-definition template as many as required, which is 3 in this case.
- ➔ To view the list of created replication controllers run the kubectl get replication controller command and you will see the replication controller listed.
- ➔ We can also see the desired number of replicas or pods, the current number of replicas and how many of them are ready. If you would like to see the pods that were created by the replication controller, run the kubectl get pods
- ➔ command and you will see 3 pods running.
- ➔ Note that all of them are starting with the name of the replication controller which is myapp-rc indicating that they are all created automatically by the replication controller.

Replica Set

It is very similar to replication controller.

As usual, first we have apiVersion, kind, metadata and spec.

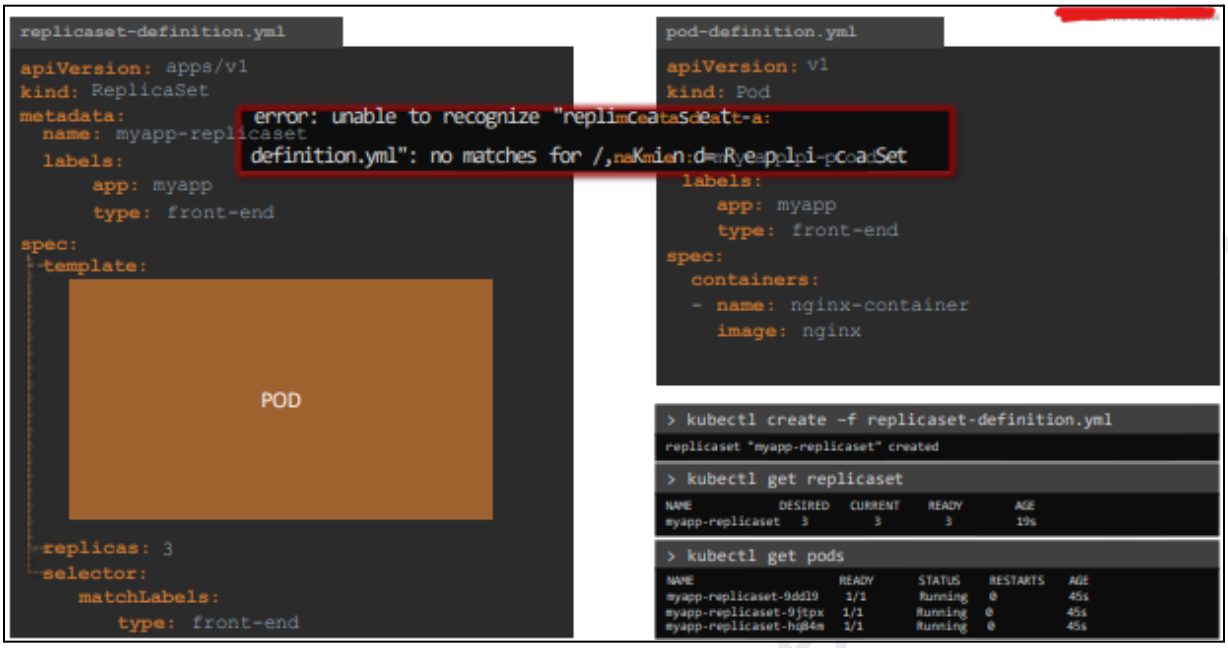
The ***apiVersion*** though is a bit different, It is "***apps/v1***" which is different from replication controller, it was simply "*v1*".

If mentioned this wrong, likely get an error as shown.
It would say "*no match for kind ReplicaSet, because the specified kubernetes api version has no support for ReplicaSet*".
The "kind" ="ReplicaSet" and add "***name***" and "***labels***" in metadata.
The specification section looks very similar to the replication controller.
It has a "***template***" section were users provide "***pod-definition***" as before. (So copy contents over from pod-definition file).
And number of "***replicas set***" to 3.
However, there is one major difference between replication controller and replica set.
Replica set requires a "***selector definition***".
The selector section helps the replicaset ***identify*** what pods fall under it.

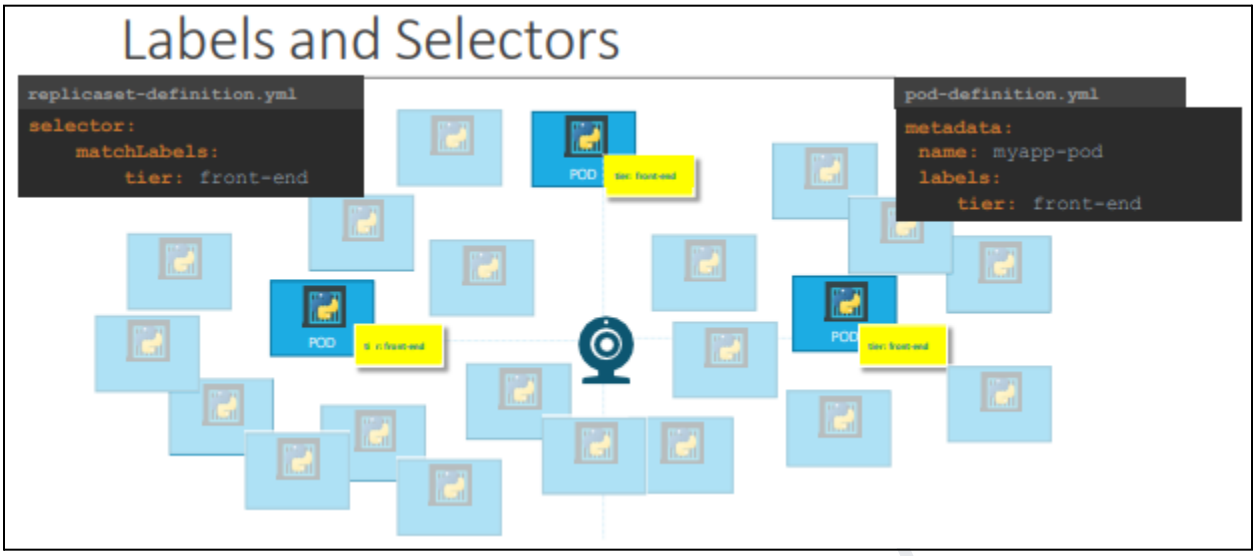
But why would you have to specify what PODs fall under it, if you have provided the contents of the pod-definition file itself in the template?

BECAUSE, replica set can ALSO manage pods that were not created as part of the replicaset creation.
For example, there were pods created BEFORE the creation of the ReplicaSet that match the labels specified in the selector, the replica set will also take THOSE pods into consideration when creating the replicas.
But before we get into that, I would like to mention that the selector is one of the major differences between replication controller and replica set.
The selector is *not a REQUIRED* field in case of a replication controller, but it is still available.
When users skip it, assume it to be the same as the labels provided in the pod-definition file.
In the case of ***replicaset*** a "*user input*" IS required for this property.
And it has to be written in the form of "***matchLabels***" as shown.
The matchLabels selector simply matches the labels specified under it to the labels on the PODs.
The replicaset selector also provides many other options for matching labels that were not available in a replication controller.
And as always to create a ReplicaSet run the kubectl create command providing the definition file as input and to see the created replicasets run the kubectl get replicaset command.
To get list of pods, simply run the kubectl get pods command.

Error received in case of:



Labels and Selectors



So what is the deal with Labels and Selectors?

Why do we label our PODs and objects in kubernetes?

Consider a simple scenario. Say we deployed 3 instances of our frontend web application as 3 PODs.

Users need to create a replication controller or replica set to ensure that they have 3 active PODs at any time.

This is one of the use cases of replica sets.

Users CAN use it to monitor existing pods, if they have them already created, as it IS in this example.

In case they were not created, the replica set will create them for users.

The role of the replicaset is to monitor the pods and if any of them were to fail, deploy new ones.

The replica set is in FACT a "***process***" that monitors the pods.

Now, how does the replicaset KNOW what pods to monitor.

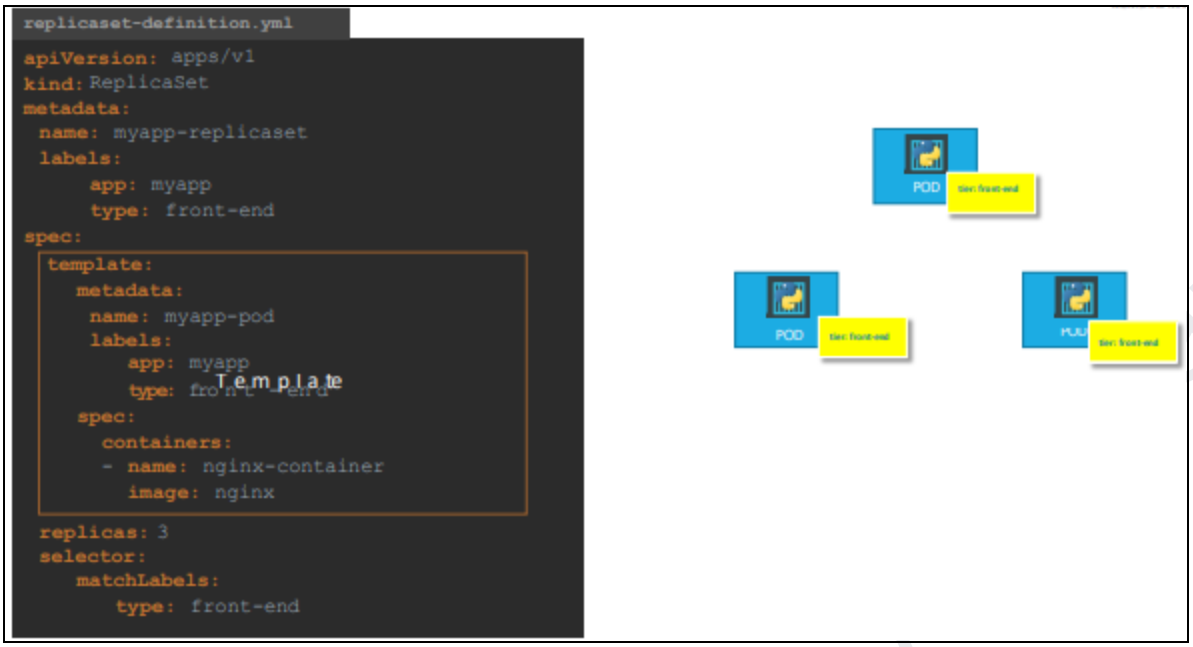
There could be 100s of other PODs in the cluster running different applications.

This is where *labeling* our PODs during creation comes in handy.

Users could now provide these labels "***as a filter***" for replicaset.

Under the selector section users use the "***matchLabels***" filter and provide the same label that we used while creating the pods.

This way the replicaset knows which pods to monitor.



Now let me ask you a question along the same lines.

In the replicaset specification section we learned that there are 3 sections: Template, replicas and the selector.

We need 3 replicas and we have updated our selector based on our discussion.

Say for instance we have the same scenario as in the previous where we have 3 existing PODs that were created already and we need to create a replica set to monitor the PODs to ensure there are a minimum of 3 running at all times.

When the ***replication controller*** is created, it is NOT going to deploy a new instance of POD as 3 of them with matching labels are already created.

In that case, do we really need to provide a template section in the replica-set specification, since we are not expecting the replicaset to create a new POD on deployment?

Yes we do, BECAUSE in case one of the PODs were to fail in the future, the replicaset needs to create a new one to maintain the desired number of PODs.

And for the replica set to create a new POD, the template definition section IS required.

SCALE

How we scale the replicaset.

Consider users started with 3 replicas and in the future decided to scale to 6.

How to update replicaset to scale to 6 replicas?

There are multiple ways to do it.

The first, is to update the number of replicas in the definition file to 6.

Then run the kubectl replace command specifying the same file using the "***-f***" parameter and that will update the replicaset to have 6 replicas.

The second way to do it is to run the ***"kubectl scale"*** command.

Use the replicas parameter to provide the new number of replicas and specify the *same file* as input.

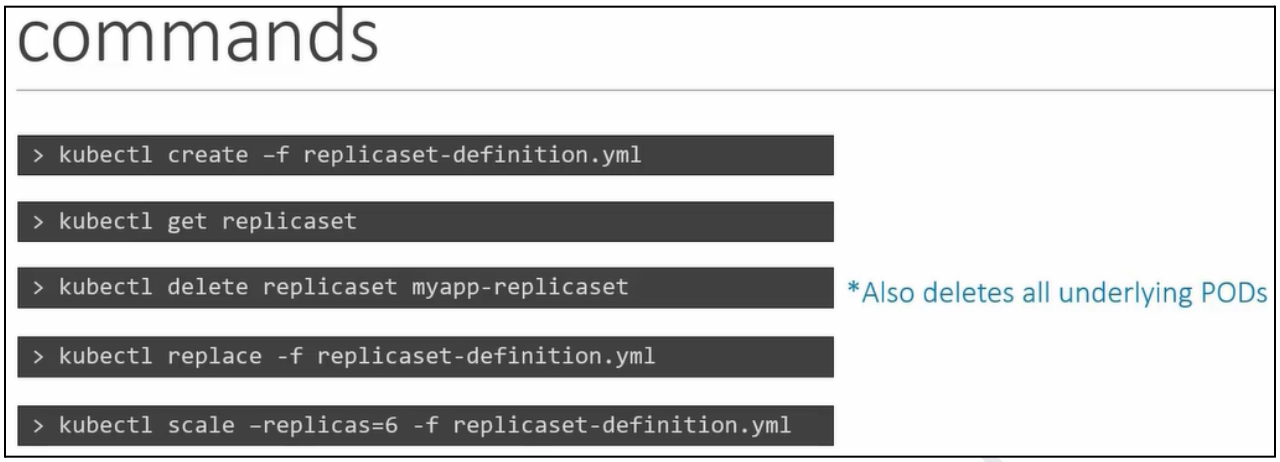
Users may either input the definition file or provide the replicaset name in the TYPE Name format.

However, Remember that using the file name as input will not result in the number of replicas being updated automatically in the file.

In otherwords, the number of replicas in the replicaset-definition file will still be 3 even though users scaled their replicaset to have 6 replicas using the kubectl scale command and the file as input.

There are also options available for automatically scaling the replicaset based on load, but that is an advanced topic.

Commands:



The kubectl create command, is used to create a replica set.
Provide the input file using the "***–f***" parameter.
Use the kubectl get command to see list of replicasets created.
Use the kubectl delete replicaset command followed by the name of the replica set to delete the replicaset. (Also deletes all underlying PODs.)
And then we have the kubectl replace command to replace or update replicaset and also the kubectl scale command to scale the replicas simply from the command line without having to modify the file.

DEMO:

replicaset.yaml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
    app: myapp
spec:
  selector:
    matchLabels:
      app: myapp
  replicas: 3
  template:
    metadata:
      name: nginx-2
      labels:
        app: myapp
    spec:
      containers:
        - name: nginx
          image: nginx
```

nginx.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-2
  labels:
    env: production
spec:
  containers:
    - name: nginx
      image: nginx
```

```
admin@ubuntu-server kubernetes-for-beginners # ls
pods  replicasets
admin@ubuntu-server kubernetes-for-beginners # cd replicasets/
admin@ubuntu-server replicasets #
```

```
admin@ubuntu-server replicasets # ls
replicaset.yaml
admin@ubuntu-server replicasets # cat replicaset.yaml
```

```
admin@ubuntu-server replicasets # kubectl create -f replicaset.yaml
replicaset.apps/myapp-replicaset created
admin@ubuntu-server replicasets #
```

```
admin@ubuntu-server replicasets # kubectl get replicaset
NAME               DESIRED   CURRENT   READY   AGE
myapp-replicaset   3         3         3       8s
admin@ubuntu-server replicasets #
```

```
admin@ubuntu-server replicasets # kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
myapp-replicaset-8nxxl   1/1     Running   0          24s
myapp-replicaset-jlgr2   1/1     Running   0          24s
myapp-replicaset-pm4rl   1/1     Running   0          24s
admin@ubuntu-server replicasets #
```

1 POD deleted but again 3 PODs are running.
Previous POD deleted and new POD is created.

```
admin@ubuntu-server replicasets # kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
myapp-replicaset-8nxxl   1/1     Running   0          45s
myapp-replicaset-jlgr2   1/1     Running   0          45s
myapp-replicaset-pm4rl   1/1     Running   0          45s
admin@ubuntu-server replicasets # kubectl delete pod myapp-replicaset-8nxxl
pod "myapp-replicaset-8nxxl" deleted
admin@ubuntu-server replicasets # kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
myapp-replicaset-bvlst   1/1     Running   0          15s
myapp-replicaset-jlgr2   1/1     Running   0          76s
myapp-replicaset-pm4rl   1/1     Running   0          76s
admin@ubuntu-server replicasets #
```

Get All details:

```
admin@ubuntu-server replicasets # kubectl describe replicaset myapp-replicaset
Name:         myapp-replicaset
Namespace:    default
Selector:     app=myapp
Labels:       app=myapp
Annotations:  <none>
Replicas:     3 current / 3 desired
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=myapp
  Containers:
   nginx:
```

TASK: Create a POD outside the Replicaset with the “same label” as Replica set and check what happens?

```
admin@ubuntu-server replicasets # cd ..
admin@ubuntu-server kubernetes-for-beginners # cd pods/
admin@ubuntu-server pods # cat nginx.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-2
  labels:
    app: myapp
spec:
  containers:
    - name: nginx
      image: nginxadmin@ubuntu-server pods #
```

```
admin@ubuntu-server pods # kubectl get pods
NAME                    READY   STATUS    RESTARTS   AGE
myapp-replicaset-bvlst  1/1     Running   0          2m44s
myapp-replicaset-jlgr2  1/1     Running   0          3m45s
myapp-replicaset-pm4rl  1/1     Running   0          3m45s
admin@ubuntu-server pods #
admin@ubuntu-server pods #
admin@ubuntu-server pods # kubectl create -f nginx.yaml
pod/nginx-2 created
admin@ubuntu-server pods # kubectl get pods
NAME                    READY   STATUS        RESTARTS   AGE
myapp-replicaset-bvlst  1/1     Running       0          2m56s
myapp-replicaset-jlgr2  1/1     Running       0          3m57s
myapp-replicaset-pm4rl  1/1     Running       0          3m57s
nginx-2                 0/1     Terminating   0          4s
admin@ubuntu-server pods #
```

```
admin@ubuntu-server pods # kubectl describe replicaset myapp-replicaset
Name:         myapp-replicaset
Namespace:    default
Selector:     app=myapp
Labels:       app=myapp
Annotations:  <none>
Replicas:     3 current / 3 desired
Pods Status:  3 Running / 1 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=myapp
  Containers:
   nginx:
    Image:        nginx
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
```

To Update the new Replicas:

```
admin@ubuntu-server replicasets #
admin@ubuntu-server replicasets # kubectl edit replicaset myapp-replicaset
```

```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this file wil
l be
# reopened with the relevant failures.
#
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  creationTimestamp: "2020-07-13T19:41:08Z"
  generation: 1
  labels:
    app: myapp
  managedFields:
  - apiVersion: apps/v1
    fieldsType: FieldsV1
    fieldsV1:
      f:metadata:
        f:labels:
          .: {}
          f:app: {}
      f:spec:
        f:replicas: {}
"/tmp/kubectl-edit-4v9li.yaml" 96L, 2490C
```

To change the number of replicas to, say, four instead of the current three, say for instance we are trying to scale up our application.

To edit the replicaset definition file and update its replicas count to *four*.

For this, we will make use of a new command called kubecti edit replicaset and we will provide the name of the replicaset, which is, *myapp-replicaset*.

Now when we run this command, we see that it opens up the running configuration of the replicaset in a text editor in a text format. In this case, it opens up in [vim].

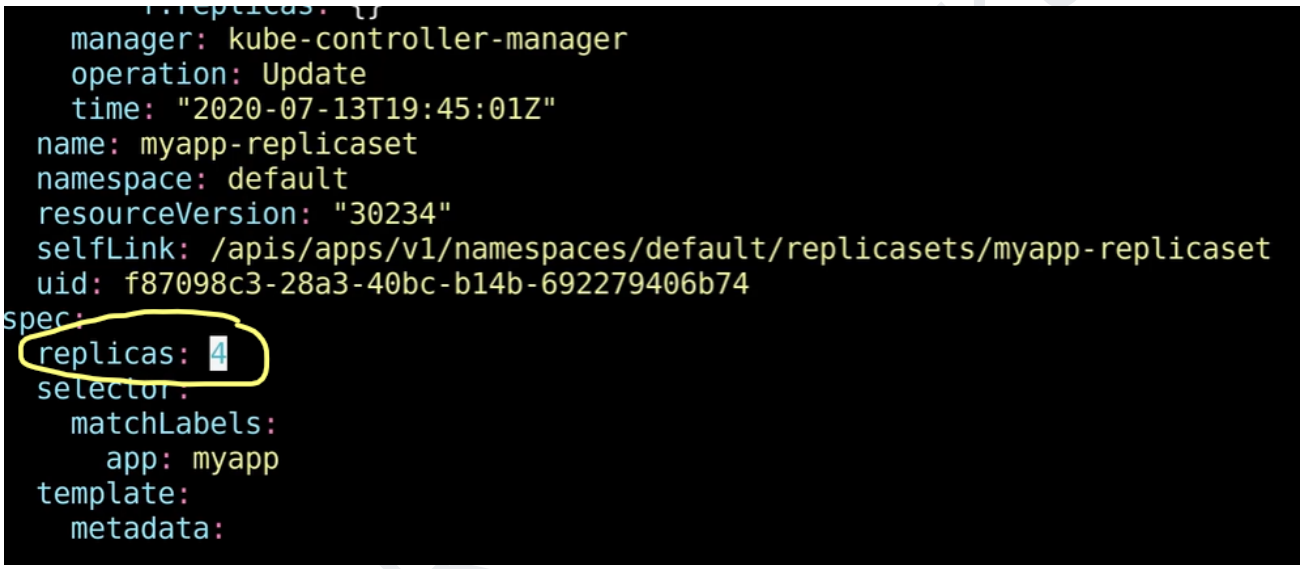
Note that this is *not the actual file* that we created at the beginning.

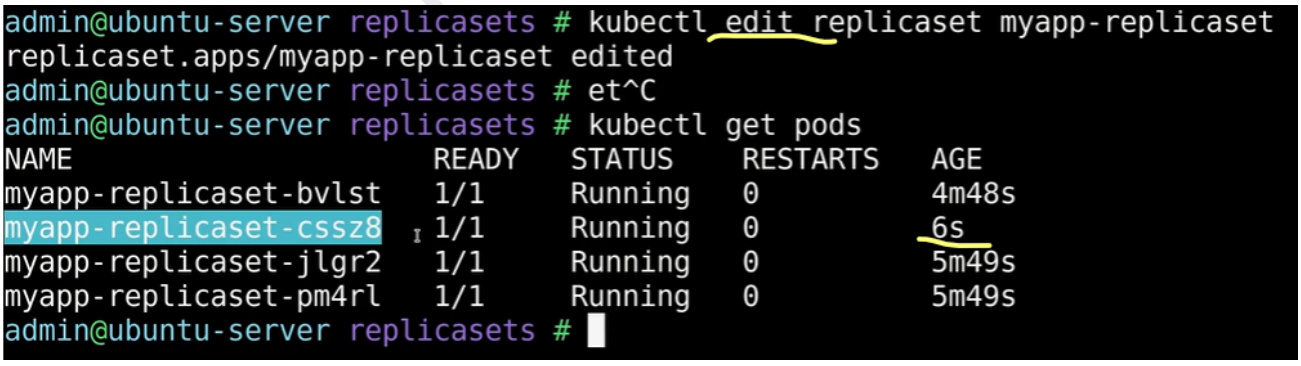
This is a temporary file that's created by Kubernetes in memory to allow us to edit the configuration of an existing object on Kubernetes of the replicaset in a text editor in a text format.

That's why you'll see a lot of additional fields in this file other than the details that users provided.

So changes made to this file are directly applied on the running configuration on the cluster as soon as the file is saved. So, be very careful with the changes that making here .

Now scroll down all the way to the ***spec section***.
which is over here, and users can see the running configuration and the current number of replicas which is set to three, now change this to scale it up by another pod, so *four* in this case, and then I will save an exit from the editor.
Now, it should now spin up a new pod in addition to the three that already had to match the new number of replicas that were specified earlier.
If users run the kubectl get pods command, now see that there is a new pod now which was spin up few seconds ago and users can use the ***same approach to scale down*** as well.





Edit with Command:

There's also a command available to scale the number of replicas without having to go in and edit the definition file and that is using the ***kubecti scale replicaset*** command.

We will provide the name of the replicaset, and we will set the number of replicas for it to scale to two. You can specify a number which is greater or less than the current number of replicas.

Take a note of the "***double dashes"*** before the replicas.

```
admin@ubuntu-server replicasets # kubectl scale replicaset myapp-replicaset --replicas
=2
replicaset.apps/myapp-replicaset scaled
admin@ubuntu-server replicasets #
```

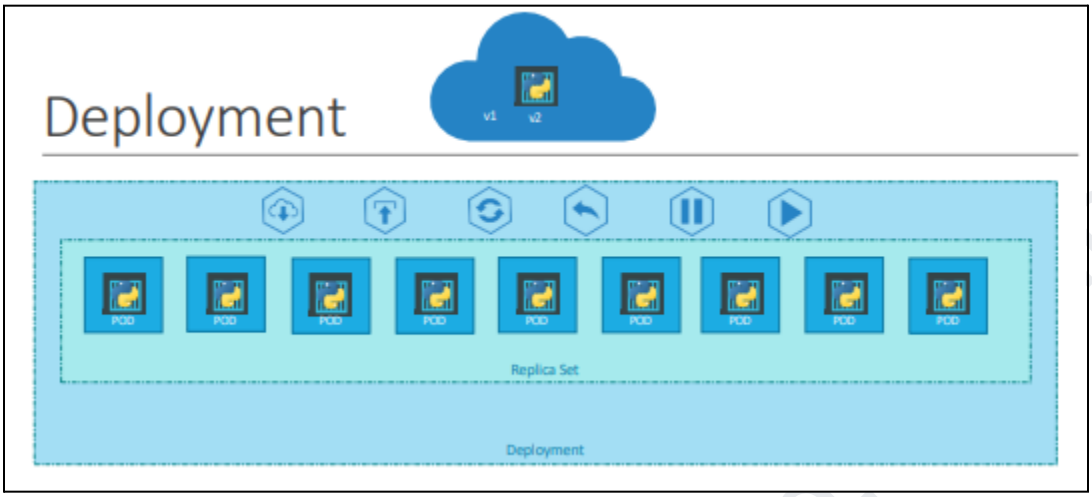
2 PODs Terminating State because mentioned 2 in previous command.

```
admin@ubuntu-server replicasets # kubectl get pods
NAME                    READY   STATUS        RESTARTS   AGE
myapp-replicaset-bvlst  0/1     Terminating   0          5m30s
myapp-replicaset-cssz8  0/1     Terminating   0          48s
myapp-replicaset-jlgr2  1/1     Running       0          6m31s
myapp-replicaset-pm4rl  1/1     Running       0          6m31s
admin@ubuntu-server replicasets #
```

Have 2 PODs only after some Time:

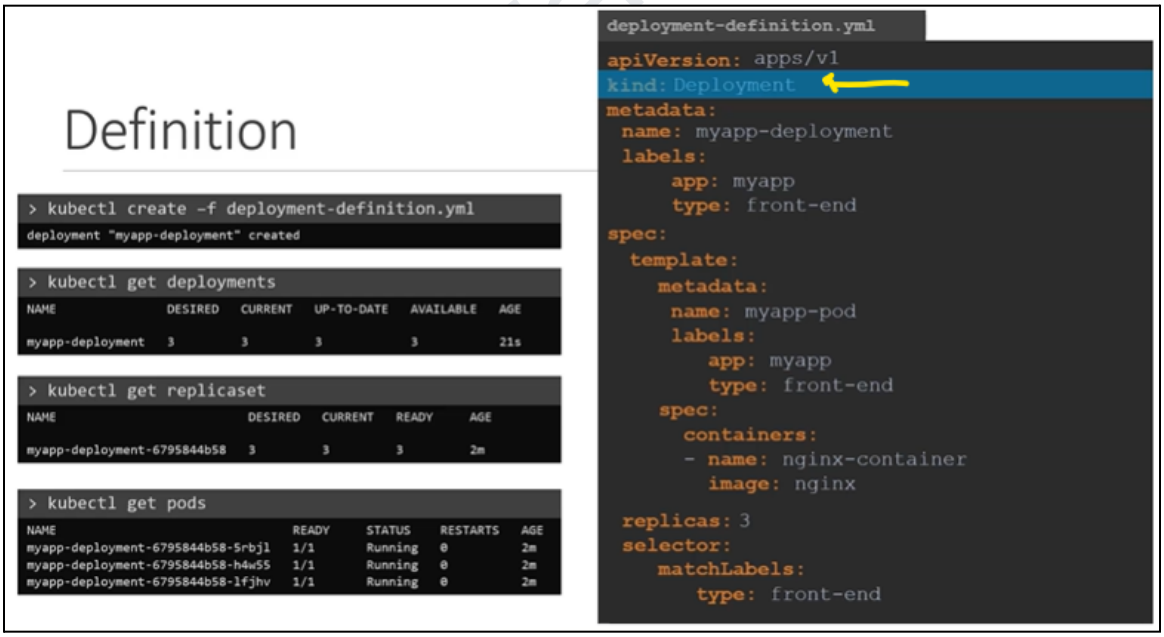
```
admin@ubuntu-server replicasets # kubectl get pods
NAME                    READY   STATUS    RESTARTS   AGE
myapp-replicaset-jlgr2  1/1     Running   0          6m50s
myapp-replicaset-pm4rl  1/1     Running   0          6m50s
admin@ubuntu-server replicasets #
```

Deployment



For a minute, let us forget about PODs and replicasets 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, when 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 our applications, so you may 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 are asked to undo the recent update.
Users would like to be able to ***rollBACK*** the changes that were recently carried out.

Finally,say for example Users would like to make multiple changes to your environment such as
upgrading the underlying WebServer versions,
as well as scaling your environment and
also modifying the resource allocations etc.
You do not want to apply each change immediately after the command is run, instead you would like to apply a pause to your environment, make the changes and then resume so that all 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 deploy 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 capabilities to upgrade the underlying instances seamlessly using rolling updates, undo changes, and pause and resume changes to deployments.



So how do we create a deployment.

As with the previous components, we first create a deployment definition file.

The contents of the deployment-definition file are exactly similar to the replicaset definition file, except for the kind, which is now going to be Deployment.

If we walk through the contents of the file it has an apiVersion which is apps/v1, metadata which has name and labels and a spec that has template, replicas and selector.

The template has a POD definition inside it.

Once the file is ready run the kubectl create command and specify deployment definition file.

Then run the kubectl get deployments command to see the newly created deployment.

The deployment automatically creates a replica set. So if you run the kubectl get replcaset command you will be able to see a new replicaset in the name of the deployment.

The replicasets ultimately create pods, so if you run the kubectl get pods command you will be able to see the pods with the name of the deployment and the replicaset.

So far there hasn't been much of a difference between replicaset and deployments, except for the fact that deployments created a new kubernetes object called deployments.

==========

Docker-vs-ContainerD

==========