Kubernetes Manifest file clarifications:

kubernetes Resource objects:

- it has a clear information about the application like image, exposing port, volume, mount path, and other credentials.
- kubernetes objects are "persistent entities" in the kubernetes system, k8s uses these entities to represent the state of your cluster.
- Once after we create the cluster, our next role is to deploy the application.
- in order to deploy the applications, we must aware of the resource objects we may need.
- k8s cluster is just an environment, rest of everything are the objects that we're going to create.
- specifically, they can describe

➔ what containerized applications are running and on which nodes they are running,

➔ the resources available to those application,

➔ The policies around how those applications behave, such as restart policies, upgrade, and fault-tolerance.

- kubernetes object is a "record of Intent", once you create the object, the k8s system will constantly work to ensure that object exists.
- There are two types of k8s object creation, modification, and deletion.

1. Imperative method

   .yaml file(kubernetes-API)

   .json (rarely used)

2. Declarative method
   kubectl CLI command (ex: kubectl apply –f …..)

YAML File:

- used to represent data, in the k8s configuration or manifest file.

KEY-VALUE pair

Fruit: Apple

Vegetable: Carrot

Liquid: Water

Array/Lists:

- collection of ordered values.

- Lists are good for ordered collections of similar items.

Fruits:

- Orange
- Apple
- Banana

Vegetables:

- Carrot
- cauliflower
- Tomato

Dictionary/Map: dict: collection of Key-value pairs.

Dictionaries are used to store information or properties of a single object as a *key-value* store format.

Dictionaries are good for representing relationships between keys and their values.

Banana:

Calories: 105

Fat: 0.4g

carbs: 27g

Grapes:

Calories: 62

Fat:0.3g

carbs:16g

Advanced Level of Same Data: List & Dictionary

Fruits:

- Banana:

  Calories: 105

  Fat: 0.4g

  carbs: 27g

- Grapes:

  Calories: 62

  Fat: 0.3g

  carbs: 16g

Dictionary Vs List Vs List of Dictionaries:

when we should use dictionary / list?

Dictionary:

Color: Blue

Model:

Name: Corvette

List:

-Grey Corvette

-Blue Corvette

-Red Corvette

year: 1995 -Green Corvette

Transmission: Manual

Price: $20,000

Dictionary with in List:

- Color: Blue

  Model:

  Name: Corvette

  year: 1995

  Transmission: Manual

  Price: $20,000

- Color: Grey

  Model:

  Name: Corvette

  year: 1995

  Transmission: Automatic

  Price: $20,000

YAML in Kubernetes:

- a kubernetes Yaml file is called ‘spec file’ or ‘manifest file’ or ‘k8s definition file’
- used as an inputs for the creation of objects such as pods, replicas, deployments, services, etc.,
- Four Top level Properties or Fields: ApiVersion, Kind, Metadata, Spec.

ApiVersion: (String value)

- Used to define the k8s API version of the Object or Resource that we're going to create.
- K8s ApiVersion is used to interpret the Resource configuration.
- Ex: apps/v1 for Deployments, StatfulSets, DaemonSets
  V1 for basic resources like Pods, Services, ConfigMaps.
  batch/V1 for Jobs, CronJobs.

Kind: (string value)

- Represents the type of Object / resource that we're going to create.
- Instructs k8s what kind of Object to create.
  - Deployment: Manages set of replicated Pods.
  - Pod: the smallest deployable unit that can run an application.
  - Service: Exposes a Set of Pods as a network service.
  - ConfigMap: Manages configuration data (as key-value pair) that can be consumed by Pods

metadata: (Dictionary format)

- Provides identifying information about the Resource or Object
- it sets resource identity attributes such as name, namespace, and labels.
- meta data helps k8s to organize and manage resources.
  - **name:** Contains unique name used to reference the object call for further processing related to the resource management and task with in the namespace.

- **namespace**: (optional) logical grouping for resources, if not specified it'd be considered as Default namespace.
- **labels:** (key-value) pairs to categorize & select resources (Ex.app: frontend) say for an example, there are hundreds of pods running a front-end application and hundreds of pods running a back-end application (or) database, it'll be difficult to group these pods once you deployed but if you label them now as 'front-end' (or) backend (or) database, you'll be able to filter the pods based on the label 'type'.
- **annotations:** (key-value) pair of metadata used to store additional, non-identifying information like 'build information'.

- it's important to note that under metadata you can only specify name or labels or anything else that k8s expects to be under metadata.
- you can't add any other property as you wish, so it's important to understand what each of these parameters expect.

spec: (dictionary format)

- contains the desired state and specific configurations for the resource.
- provides detailed instructions on how the resource should behave.

  ex: -no.of replicas for a Deployment,
  -container images for Pods,

-Load Balancing settings for Services.

- spec defines what's inside the object that we are creating.

Practical Knowledge:

1) If the image state is "waiting /error ", what could be the reason? we have to find the reason in "Events" section. if the reason is "Image Pull Back Off", that means the image is not available in our Docker Hub or in the private registry from where we're pulling.
2) Create a New Pod "redis" and image "redis123", use a "pod-definition" yaml file, yes the image name is wrong take it for example.
   - ➔ kubectl run redis - -image =redis123 - - dry_run=client –o yaml>redis_definition.yaml (to preview)
   - ➔ kubectl create –f redis_definition.yaml
3) How to edit the running pod ?
   - ➔ kubectl edit pod <podname> (used for minor change.)
   - ➔ kubectl apply –f <file name>.yaml> (do the changes in the manifest file and execute this command, used for long term or larger updates.)

Limitations:

- not all fields can be modified in a running pod.
- pods are generally immutable for fields like "containers", so changes to image versions, environment variables, and other configuration options may not apply to the existing pod.

Restart Requirement:

- you have to delete and recreate the pod if we had to edit the pod in some cases.
- or edit the deployment (or other controller) managing the pod to apply the changes.

Auto Save:

- for "kubectl edit command" after making changes, saving & closing the editor will automatically apply the changes to the pod.

  but it is always recommended to "Turn off" the "Auto Save" to avoid "Premature apply"

Protocol to Edit the Pod :

1) Keep your favorite Editing tool/terminal as default.
2) -> kubectl edit pod –o yaml> my-pod.yaml
3) kubectl apply –f my-pod.yaml

kubctl run Vs create Vs apply ?

kubectl run > creates a pod for quick testing, debugging, (temporary pods)

kubectl create > creates a resource /object from the manifest file we write.

kubectl apply> creates or update resources (Declarative, version-controlled Management)

4)How to deploy application using declarative (kubectl) command?

→ kubectl run httpd - -image=httpd --replicas=1 --port=80

but it is not recommended to do it in real time, since we can't edit once we deploy, so we can't keep them in VCS.

Resources:

RepicaSet: (deprecated unofficially)

- Ensures that the specified number of pods are running as directed by the Deployment**.**
- Deployments use ReplicaSets as building block to manage the pods.
- ReplicaSet responsible for maintaining the availability and scalability of the applications by managing the lifecycle of pods.
- can be used independently, but managed by Deployments which provides additional features for application life cycle management.

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: myapp-deployment
    labels:
        app: myapp
        type:  front-app
spec:
```

```
template:
    metadata:
        name: myapp-pod
        labels:
            app: myapp
            type: front-end
    spec:
        containers:
        -   name: nginx-container
            image: nginx
replicas: 3
selector:
    matchLabels:
        type: front-end
```

Pod template (pod spec file)

- if we use Deployments in k8s, we don't need to manage ReplicaSet separately, as per the Deployment Life Cycle it manages ReplicaSet behind the screen.
- Deployment is more versatile than ReplicaSet, so it is always recommended to use Deployment in Prod environment in real time.
- "template" section we create under Spec to provide is a "pod template" to be used by the "Deployment".

How do we define Pod template?

- it is nothing but the object “Pod” manifest or definition files.
- we can keep “metadata” and “spec” definition of the Default pod manifest file.
- “template” section is a like child of main “spec” property.
- “metadata” and “spec” those we mention with in the “template” is like a child of “template” sub-property.
- so we have to define ‘metadata’, one is for main ‘Resource’ and another one is for ‘template’ sub-property. likewise, we have two define “spec” file one for each.
- “replicas” to determine the no of pods.
- “selector” to determine which pod or pods the main Resource(Deployment) going to manage.
- it allows k8s to match the pods created by Deployment with the desired specifications set in the file.

while we have ‘template’ of pod already why do we need selector section?

- it’s because ReplicaSet can also manage pods that we’re not created as part of the ReplicaSet creation. for example, there were some pods created before even we create ReplicaSet, that matches “labels” specified in the “Selector”
- the ReplicaSet will also take those pods into consideration when creating the replicas.
- as of now we define ‘Selector’ in Deployment which manages behind the screen

matchLabels:

- the ‘matchLabels’ sub-property simply matches the labels specified under it in the labels on the pod.

so what is the deal with ‘Label’ and ‘Selector’? why do we labeling our objects in k8s?

- Labels are ‘key-value’ pair that you attach to objects to provide metadata that is flexible, efficient**,** and easily searchable.
- “Labels” make it possible to group, identify and manage k8s resources in powerful way.
- we provide labels to the Deployment as a filter to identify & choose the required pods.
- under the ‘Selector’ section we use the ‘match labels’ filter and provide the same label that we used while creating the pod.
- when the Deployment is created, it is not going to create new instances of pods as ‘three’ of them matching labels are already created.

In that case do we really need to provide a ‘template’ section in the Deployment or ReplicaSet since we’re not expecting the Replicas to create a new pod on Deployment?

yes, we need to keep the template section despites the scenario. Because if one of the pods get failed in future, the ‘Deployment’ would create a new one using the ‘template’ section only.

what are the reasons for Labeling objects in k8s?

1. **Grouping and selection:**
   - labels allow you to logically group related resources without relying on rigid hierarchies.

- we can use “Label Selectors” to target specific groups of resources based on their labels.
  for example- for instance, a ‘Service’ can use a ‘label selector’ to route traffic only to the pods with a certain label (ex: app: front-end).

1) Oraganization & Filtering

- Labels helps you to organize resources by their purpose, environment, or other distinguishing attributes.
  (ex; env: production or tier: back-end
- you can filter resources using labels to quickly find or manage only the relevant ones, even in large clusters with many resources.

2) **Scaling & Updates**

- k8s uses labels to manage pod replication through ReplicaSet or Deployment, by labeling pods, k8s can select and scale the suitable ones.
- During “Rolling Updates”, Deployment uses Labels to differentiate between the old and New version of pods

3) **Monitoring and Analytics**

- Labels enable monitoring and logging tools (like Prometheus and Grafana) to categorize and display metrics based on specific criteria.
  Ex; you can monitor only the **app: front-end** pods across environments.

4) **Automation & CICD Pipelines:**

- labels support automation by providing a way to dynamically group and manage resources during CICD pipeline workflows.
- you can use labels to trigger automated tasks, like only updating pods with (**env: staging)** during a test environment.

Related commands to Deployment or ReplicaSet:

➔ **kubectl get replicaset or rs**

➔ **kubectl describe rs < rs name>** here we could see the events which has all history including the deleted replicasets.

➔ **kubectl edit rs <rs name>**

➔ **kubectl delete replicaset < replicaset name>**

➔ **kubectl get rs <rs name> -o yaml** to display manifest file of the particular replicaset or any other resources of k8s.

How do we scale our Application?

1) **Updating the Deployment.yaml file:**
   so modify **replica: 6**
2) **Through 'kubectl' command line utility tool**
   ➔ **kubectl replace –f <file name>**
3) **Scale-File Input**
   ➔ **kubectl scale - -replicas=6 –f <file name>**
   it is not an automatic update format. we've to do it manually.

4) **Type-Name format**

➔ **kubectl scale <type name> - -replicas=6 <file name>**

we can also automatically be scaling the Deployment based on the load, by using the advanced techniques.

what happens if we create a new pod outside of the ReplicaSet or Deployment, with the same "label" which Deployment or ReplicaSet has?

it'll be automatically terminated before even created, because it's not allowing to create more pod with the same label, since the replicaset has desired state to keep.

Lab knowledge:

1) **if the pod is not 'ready' what we need to check?**
   - **check the status through 'get pods' command**
   - **check the Events through 'describe' command**
   - **if the error message is "ImagePullBackOff" means the image is not existing in the Registry from where we're pulling.**
2) **what are the things that we need to be aware while writing a 'ReplicaSet' or 'Deployment'?**
   - **have to choose the right 'apiVersion' (ex; app/v1)**

   ➔ **kubectl api-resources | grep replicaset**
   - **Labels & matchLabels should match.**

Deployment:

The deployment provides us with the capability to upgrade the underlying instances seamlessly using the "Rolling update", Roll back" and Pause and Resume changes as required.

Deployment Related commands:

➔ **kubectl create –f <deployment file name>**

➔ **kubectl get deployments**

➔ **kubectl describe deployment < deployment file name>**

attributes:

Name, Namespace, CreationTimeStamp, Labels, Annotations, Selector, Replicas, Strategy Type, MinReadySeconds, RollingUpdateStrategy, PodTemplate, Labels, containers, nginx, Image, port, HostPort, Environment, Mounts, Volumes, conditions, OldReplicaSets, NewReplicaSet.

➔ kubectl rollout status deployment/<deployment name>

➔ kubectl rollout history deployment/<deployment name>

➔ kubectl get deploy deployment-nginx –o yaml

➔ rm –rf <.yaml file name> - for deleting the file, if you had applied the deleted .yaml file already, you can get the .yaml file again through deploy edit command. just you need to remove the system generated annotations.

How to create a Deployment file?

- create a new directory called "Deployment" under your project directory, and create new file 'deployment.yaml' file.

VS-code tool:

- we can have ‘split screen’ feature to take contents and work multiple files together.

Linux Knowledge:

To find and replace the word:

➔ **:% s/old_word/new_word/g**
➔ **:%s/old_word/new_word/gc**
➔ **:10,20s/old_word/new_word/g**

:%s = specifies to search(s), and replace (% for the entire file)

g = (globally) replace all occurrences on each line.

c = this will prompt you to confirm each time before replacing

10,20 = from 10th to 20th line.

Update and Roll back in Deployment:

Rollout and Versioning:

- before we look into how to upgrade our application, let’s try to understand rollout and versioning in a ‘Deployment’
- when you first create a Deployment, it triggers a ‘Roll out’,
- A new ‘Roll out’ creates a new deployment ‘Revision’ (Revision 1)
- in the future when the application is upgraded, meaning when the container version is updated to a new one, a New “Roll out” is triggered and a new ‘Deployment Revision’ is created named (Revision 2).
- This helps us to keep track of the changes made to our deployment and enables us to Roll back to a previous version if necessary.