Linux Academy

Certified Kubernetes Application Developer (CKAD) Study Guide

Will Boyd

willb@linuxacademy.com

March 28, 2019

Contents

Core Concepts

Kubernetes API Primitives

Documentation:

- Kubernetes Objects

Kubernetes API Primitives are called “Kubernetes Objects” in the documentation.

They are data objects that define the state of the cluster. Each object has a spec and a status. * Spec - defines the desired state. * Status - describes the current state.

`kubectl get` returns a list of object types available to the cluster.

Define an object’s spec in the form of yaml data, for use for creating and modify objects.

Obtain the spec and status with commands like `kubectl describe`:

```
kubectl describe $object_type $object_name
```

You can also get information about an object with `kubectl get`:

```
kubectl get $object_type $object_name
```

Add the `-o yaml` flag to get the data in yaml format.

Creating Pods

Documentation:

- Pod Overview

Pod: A collection of one or more containers and their shared resources.

One way to create Kubernetes objects, such as pods, is to define the object spec in a yaml file. For example:

`my-pod.yml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
```

```
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
```

Create an object from a yaml definition file with `kubectl create -f my-pod.yml`.

After an object is created, alter it by changing the yaml file and using `kubectl apply -f my-pod.yml`.

Editing objects directly is possible, with `kubectl edit $object_type $object_name`.

Namespaces

Documentation:

- Namespaces

Most Kubernetes objects reside in namespaces.

Assign an object to a specific namespace using the object metadata.

For example, `my-pod.yml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  namespace: my-namespace
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
```

When working with objects using `kubectl`, specify the namespace with the `-n` flag:

```
kubectl get pods -n my-namespace
```

Whenever a namespace is not specified, the cluster will assume the `default` namespace.

Basic Container Configuration

Documentation:

- Define a Command and Arguments for a Container

There are various ways of configuring containers within a Pod specification:

Use `command` to specify the command that will be used to execute the container:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-command-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['echo']
  restartPolicy: Never
```

Use `args` to specify any custom arguments that will be used to execute the container:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-args-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['echo']
    args: ['This is my custom argument']
  restartPolicy: Never
```

Use `containerPort` to expose ports to the cluster:

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: my-containerport-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: nginx
    ports:
    - containerPort: 80
```

Configuration

ConfigMaps

Documentation:

- Configure a Pod to Use a ConfigMap

ConfigMap: Kubernetes Object that contains key-value data for use in configuring containers:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config-map
data:
  myKey: myValue
  anotherKey: anotherValue
```

Add ConfigMap data to a pod as an environment variable:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-configmap-pod
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', "echo $(MY_VAR) && sleep 3600"]
    env:
    - name: MY_VAR
      valueFrom:
        configMapKeyRef:
```

```
          name: my-config-map
          key: myKey
```

Add ConfigMap data to a pod as a mounted volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-configmap-volume-pod
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'ls /etc/config && sleep 3600']
    volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: myConfigMap
```

SecurityContexts

Documentation:

- Configure a Security Context for a Pod or Container

Use a pod's `securityContext` to specify particular OS-level privileges and permissions for a pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-securitycontext-pod
spec:
  securityContext:
    runAsUser: 2000
    fsGroup: 3000
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', "echo Hello Kubernetes! && sleep 3600"]
```

Resource Requirements

Documentation:

- Resource requests and limits of Pod and Container
- Resource request: The amount of resources a container needs to run - Kubernetes uses these values to determine whether or a not a worker node has enough resources available to run a pod.
- Resource limit: The maximum resource usage of a container - The container run time will try to prevent the container from exceeding this amount of resource usage.

You can specify resource requests and limits in the container spec:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-resource-pod
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

Secrets

Documentation:

- Secrets

Secret: A Kubernetes object that stores sensitive data, such as password, keys, or tokens.

Create a secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
stringData:
  myKey: myPassword
```

Use secret data in a container as an environment variable:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-secret-pod
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', "echo Hello, Kubernetes! && sleep 3600"]
    env:
    - name: MY_PASSWORD
      valueFrom:
        secretKeyRef:
          name: my-secret
          key: myKey
```

ServiceAccounts

Documentation:

- Managing Service Accounts
- Configure Service Accounts for Pods

ServiceAccounts are used for allowing pods to interact with the Kubernetes API, and for controlling what those pods have access to do using the API. Specify the service account that a pod will use, when interacting with the Kubernetes API, using the `serviceAccountName` attribute in the pod spec:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-serviceaccount-pod
spec:
  serviceAccountName: my-serviceaccount
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', "echo Hello, Kubernetes! && sleep 3600"]
```

Multi-Container Pods

Understanding Multi-Container Pods

Documentation:

- Using a sidecar container with the logging agent
- Communicate Between Containers in the Same Pod Using a Shared Volume
- The Distributed System ToolKit: Patterns for Composite Containers

Multi-container pods are pods that have more than one container. Creating them is as simple as listing multiple containers under the `containers` section of the pod spec.

Containers within a pod can interact with each other in various ways:

- Network: Containers can access any listening ports on containers within the same pod, even if those ports are not exposed outside the pod.
- Shared Storage Volumes: Containers in the same pod can be given the same mounted storage volumes, which allows them to interact with the same files.
- Shared Process Namespace: Process namespace sharing can be enabled by setting `shareProcessNamespace` `true` in the pod spec. This allows containers within the pod to interact with, and signal, one another's processes.

Some common design patterns for multi-container pods are:

Ambassador: An haproxy ambassador container receives network traffic and forwards it to the main container. ***Example:*** An ambassador container listens on a custom port, and forwards the traffic to the main container's hard-coded port.

A concrete example would be a configmap storing the haproxy config. Haproxy will listen on port 80 and forward the traffic to the main container, which is hard-coded to listen on port `8775`:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: haproxy-sidecar-config
data:
  haproxy.cfg: |-
    global
        daemon
        maxconn 256

    defaults
```

```
        mode http
        timeout connect 5000ms
        timeout client 50000ms
        timeout server 50000ms

    listen http-in
        bind *:80
        server server1 127.0.0.1:8775 maxconn 32
```

Pod definition for the pod, which implements the ambassador model:

```
apiVersion: v1
kind: Pod
metadata:
  name: fruit-service
spec:
  containers:
  - name: legacy-fruit-service
    image: linuxacademycontent/legacy-fruit-service:1
  - name: haproxy-sidecar
    image: haproxy:1.7
    ports:
    - containerPort: 80
    volumeMounts:
    - name: config-volume
      mountPath: /usr/local/etc/haproxy
  volumes:
  - name: config-volume
    configMap:
      name: haproxy-sidecar-config
```

Sidecar: A sidecar container enhances the main container in some way, adding functionality to it. ***Example:*** a sidecar periodically syncs files in a webserver container's file system from a Git repository.

Adapter: An adapter container transforms the output of the main container. ***Example:*** An adapter container reads log output from the main container and transforms it.

A concrete example is a main container mounting `/var/log` as a shared volume. An adapter container running fluentd also mounts this same volume, and is able to read all log output from the main container. A configmap provides the configuration for fluentd. Fluentd reads the main container's logs, transforms them, and writes the transformed output to a separate storage volume.

Fluentd config, stored in a ConfigMap:

```
apiVersion: v1
kind: ConfigMap
```

```
metadata:
  name: fluentd-config
data:
  fluentd.conf: |
    <source>
      type tail
      format none
      path /var/log/1.log
      pos_file /var/log/1.log.pos
      tag count.format1
    </source>

    <source>
      type tail
      format none
      path /var/log/2.log
      pos_file /var/log/2.log.pos
      tag count.format2
    </source>

    <match **>
      @type file
      path /var/logout/count
      time_slice_format %Y%m%d%H%M%S
      flush_interval 5s
      log_level trace
    </match>
```

Pod config, implementing the adapter model:

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args:
    - /bin/sh
    - -c
    - >
      i=0;
      while true;
      do
        echo "$i: $(date)" >> /var/log/1.log;
        echo "$(date) INFO $i" >> /var/log/2.log;
```

```
        i=$((i+1));
        sleep 1;
      done
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  - name: count-agent
    image: k8s.gcr.io/fluentd-gcp:1.30
    env:
    - name: FLUENTD_ARGS
      value: -c /etc/fluentd-config/fluentd.conf
    volumeMounts:
    - name: varlog
      mountPath: /var/log
    - name: config-volume
      mountPath: /etc/fluentd-config
    - name: logout
      mountPath: /var/logout
  volumes:
  - name: varlog
    emptyDir: {}
  - name: config-volume
    configMap:
      name: fluentd-config
  - name: logout
    hostPath:
      path: /home/cloud_user/log_output
```

Observability

Liveness and Readiness Probes

Documentation:

- Container probes
- Configure Liveness and Readiness Probes

Liveness probe: Determines whether the container is running properly - When a liveness probe fails, the container will be shut down or restarted, depending on its RestartPolicy.

Readiness Probe: Determines whether the container is ready to serve requests - Requests will not be forwarded to the container until the readiness probe succeeds.

Create liveness and readiness probes including them in the pod spec.

Here is a liveness probe that runs a command in order to determine whether the container is running properly:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-liveness-pod
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', "echo Hello, Kubernetes! && sleep 3600"]
    livenessProbe:
      exec:
        command:
        - echo
        - testing
      initialDelaySeconds: 5
      periodSeconds: 5
```

This pod includes a readiness probe that makes an http request to determine readiness:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-readiness-pod
spec:
  containers:
  - name: myapp-container
    image: nginx
    readinessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 5
```

Container Logging

Documentation:

- Logging Architecture

Access container logs using the `kubectl logs` command:

```
kubectl logs <pod name> -c <container name>
```

Note: If the pod only has one container, omit the `-c <container name>`.

Manipulated and/or redirect the output of `kubectl logs` to a file using normal command-line techniques:

```
kubectl logs <pod name> > /path/to/output.log
```

Monitoring Applications

Documentation:

- Tools for Monitoring Resources

List resource usage for all pods in the default namespace:

```
kubectl top pods
```

List resource usage for a specific pod:

```
kubectl top pod resource-consumer-big
```

For pods in a specific namespace:

```
kubectl top pods -n kube-system
```

Get resource usage for nodes:

```
kubectl top nodes
```

Debugging

Documentation:

- Troubleshoot Applications
- Debug Pods and ReplicationControllers
- Debug Services

To debug in Kubernetes and locate a problem, you need to know how to explore the cluster and find information about objects. Then you'll need to know how to edit objects in order to fix the problem.

kubectl get

Use `kubectl get` to list objects, for example: `kubectl get pods`.

You can also list other object types, i.e. `kubectl get deployments`, `kubectl get services`.

`kubectl get` provides the `STATUS` and `READY` information for pods, a good way to spot problems!

Don't forget to use the `-n` flag to explore different namespaces, because the problem may not be in the default namespace. If you're not sure what namespaces are present in clusters, do `kubectl get namespaces`.

You can also use the `--all-namespaces` flag, like this: `kubectl get pods --all-namespaces`. It lists objects from all namespaces, which is very useful for finding problems quickly if you don't know what namespace to look in.

kubectl describe

Use `kubectl describe` to get more information on a specific object. If you see a pod with a bad `STATUS`, do a `kubectl describe pod <pod name>` on that pod to find out what is wrong.

kubectl logs

Sometimes something goes wrong inside the container, and `kubectl describe` does not provide enough information to find out what is wrong. Use `kubectl logs <pod name>` to get the container logs.

If the pod has multiple containers, you will need to specify which container to get logs from with `kubectl logs <pod name> -c <container name>`.

kubectl edit

Once you find an object with a problem, you may need to edit the object. You can use `kubectl edit` to make changes to the object using the default editor, like this: `kubectl edit pod <pod name>`.

Export object yaml

`kubectl edit` may not work in all situations, such as when you need to change values that can't be edited once the object is initialized. In these cases, you may need to delete and re-create the object. It is a good idea to back up the yaml definition before deleting the object, so that you can fix it and then re-create it later. Do this with the `-o yaml --export` flags: `kubectl get <object type> <object name> -o yaml --export`.

kubectl apply

When there is a yaml descriptor file for the object, this changes the file and re-applies it: `kubectl apply -f <file>`.

Pod Design

Labels, Selectors, and Annotations

Documentation:

- Labels and Selectors
- Annotations

Label: Key-value object metadata used to identify, select, and group objects

Apply labels using the `metadata.labels` attribute:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-production-label-pod
  labels:
    app: my-app
    environment: production
spec:
  containers:
  - name: nginx
    image: nginx
```

Selector: Used to select a group of objects using labels

Use selectors with `kubectl get -l` to get selected objects. For example:

```
kubectl get -l app=my-app
```

Equality-based selectors: `app=my-app`, `environment!=production`

Set-based selectors: `environment in (development,production)`

Chain multiple selectors in a comma-delimited list: `app=my-app,environment=production`

Annotation: Key-value object metadata, but cannot be used to identify objects and cannot be used in a selector

Apply annotations using the `metadata.annotations` attribute:

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: my-annotation-pod
  annotations:
    owner: terry@linuxacademy.com
    git-commit: bdab0c6
spec:
  containers:
  - name: nginx
    image: nginx
```

Deployments

Documentation:

- Deployments

Deployment: Defines a desired state for a set of replica pods, and works to maintain that state by creating, removing, and modifying those pods

An example of a deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

- **spec.replicas:** The number of replica pods
- **spec.template:** A template pod descriptor which defines the pods which will be created
- **spec.selector** The deployment will manage all pods whose labels match this selector. When creating a deployment, make sure the selector matches the pods on the template!

Rolling Updates and Rollbacks

Documentation:

- Updating a Deployment
- Rolling Back a Deployment

Rolling update: Gradually rolling out a change to a set of replica pods to avoid downtime

This is how you can perform a rolling update using a deployment:

```
kubectl set image deployment/<deployment name> <container name>=<image name> --record
```

This sets the image for the deployment to the value specified for `<image name>`.

`--record` records the changes made during the rolling update. This data can be used later to roll back the change.

Rollback: Reverting to a previous state after an update

Get a list of previous rolling updates like so:

```
kubectl rollout history deployment/<deployment name>
```

Use the `--revision` flag to get more information on a specific revision:

```
kubectl rollout history deployment/<deployment name> --revision=<revision number>
```

You can rollback to the state before last revision with `kubectl rollout undo`:

```
kubectl rollout undo deployment.v1.apps/<deployment name>
```

Or use the `--to-revision` flag to roll back to a specific earlier revision:

```
kubectl rollout undo deployment.v1.apps/<deployment name> --to-revision=<revision number>
```

You can control certain aspects of the rollingUpdate strategy in the deployment spec:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  strategy:
    rollingUpdate:
      maxSurge: 3
      maxUnavailable: 2
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

spec.strategy.rollingUpdate.maxSurge: This represents the maximum number of extra replicas (above the deployment's normal replica count) that can be created, at a time, during a rolling update. **spec.strategy.rollir** This sets the maximum number of replicas that can be unavailable, at a time, during a rolling update.

Jobs and CronJobs

Documentation:

- Jobs - Run to Completion
- CronJob
- Running Automated Tasks with a CronJob

Job: Creates one or more pods to do work and ensures that they successfully finish

This job calculates 2000 digits of pi. When it is done, the container will exit:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
```

```
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

You can use `kubectl get` with Jobs:

```
kubectl get jobs
```

CronJob: Executes jobs on a schedule

This CronJob runs the job specified by the `jobTemplate`, according to the schedule specified by the cron expression in `schedule`:

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

You can use `kubectl get` with CronJobs:

```
kubectl get cronjobs
```

Services and Networking

Services

Documentation

- Services
- Using a Service to Expose Your App

Service: An abstraction layer which provides network access to a dynamic, logical set of pods

An example of a service:

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  type: ClusterIP
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 8080
    targetPort: 80
```

- type: Sets the service type
- selector: Selector used to determine which pods are included in the service
- ports.port: The port that the service listens on
- ports.targetPort: The port which traffic is forwarded to on the pods

Service types

- ClusterIP: Service is exposed within the cluster using its own IP address, and can be located via the cluster DNS using the service name
- NodePort: Service is exposed externally on a listening port on each node in the cluster
- LoadBalancer: Service is exposed via a load balancer created on a cloud platform:
    - The cluster must be set to work with a cloud provider in order to use this option.

- ExternalName: Service does not proxy traffic to pods, but simply provides DNS lookup for an external address:
    - This allows components within the cluster to look up external resources in the same way they look up internal ones: through services.

NetworkPolicies

Documentation:

- Network Policies

NetworkPolicy: Uses label selectors to select pods and define network access rules around those pods

A sample NetworkPolicy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
spec:
  podSelector:
    matchLabels:
      app: MyApp
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
```

```
ports:
- protocol: TCP
  port: 5978
```

- podSelector: Defines the selector, which determines which pod the NetworkPolicy applies to
- policyTypes: Can contain `Ingress`, `Egress`, or both:
    - This determines which type(s) of traffic the policy applies to, whether it's incoming or outgoing traffic.

Ingress rules

Ingress rules provide a whitelist for traffic coming in to the pod. Any traffic that does not match any of the ingress entries will be blocked. Ingress rules specify a list of sources under `from`, as well as one or more `ports`. In the above example, any traffic coming in to port `6379` on the pod would be allowed, as long as it matches one of the 3 listed sources. All other traffic would be blocked.

Egress rules

Egress rules provide a whitelist for traffic coming out of the pod. Egress rules specify a list of destinations under `to`, as well as one or more `ports`. In the above example, egress traffic will only be allowed if it going to port `5978`, and to an IP address that falls within the specified CIDR range.

To and From Selectors

Ingress and Egress rules use the same syntax to specify sources and destinations for traffic, under `from` and `to`, respectively.

- podSelector: This allows for specifying a pod selector. Traffic from/to pods matching the selector will match the rule.
- namespaceSelector: This is similar a podSelector, but allows you to select one or more namespaces instead. Traffic from any pods in the matched namespaces will match the rule. ***However,*** if you use podSelector and namespaceSelector together, the matched pods must also reside in one of the matched namespaces.
- ipBlock: This allows you to specify a range of IP addresses, using CIDR notation. Sources/destinations that fall within the IP range will match the rule. You can also list IPs to exclude from the range using `except`.

State Persistence

Volumes

Documentation:

- Volumes

Volumes: These provide storage to containers that is outside the container, and can therefore exist beyond the life of the container. Containers in a pod can share volumes, allowing them each to interact with the same files.

A sample container with an `emptyDir` volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-pod
spec:
  containers:
  - image: busybox
    name: busybox
    volumeMounts:
    - mountPath: /tmp/storage
      name: my-volume
  volumes:
  - name: my-volume
    emptyDir: {}
```

emptyDir: Volumes are automatically deleted when the pod is removed from a node. They can be an easy way to allow two containers to share storage, without need for more complex storage solutions.

PersistentVolumes and PersistentVolumeClaims

Documentation:

- Persistent Volumes
- Configure a Pod to Use a PersistentVolume for Storage

PersistentVolumes

PersistentVolume: An object which represents a storage resource available to the cluster - Just like a Node represents CPU and memory resources, a PV (PersistentVolume) represents a storage resource.

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: my-pv
spec:
  storageClassName: local-storage
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  local:
    path: "/mnt/data"
```

- storageClassName: Defines the type of storage the PV represents
    - It can be used to create multiple types of storage resources for different needs such as fast, slow, short-term, long-term, etc.
- capacity: The amount of storage represented by this PV
- accessModes - Determines the manner in which the resource is mounted to containers
    - ReadWriteOnce means that only one container can mount the resource in read/write mode.
- type: The above PV uses a `hostPath` type, which simply uses a directory on a node for storage. Kubernetes supports many types of PVs, such as nfs, StorageOS, etc.

PersistentVolumeClaims

PersistentVolumeClaim: A request for a storage resource, PVCs (PersistentVolumeClaims) provide an abstraction layer between users (Pods) and Storage:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my-pvc
spec:
  storageClassName: local-storage
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 512Mi
```

- storageClassName: This specifies the storage class requested by the claim. The PVC itself will be bound to a PV with a matching `storageClassName` and one or more matching `accessModes`. If no such PV exists, the PVC will remain unbound.

- accessModes: These are the access modes requested by the PVC.
- resources.requests.storage: This a resource request that specifies how much storage is required by the PVC.

Use a PersistentVolumeClaim in a Pod

You can utilize storage resources in a Pod by creating a `volume` that references a PVC, and mounting it to a container:

```
kind: Pod
apiVersion: v1
metadata:
  name: my-pvc-pod
spec:
  volumes:
  - name: my-storage
    persistentVolumeClaim:
      claimName: my-pvc
  containers:
  - name: busybox
    image: busybox
    command: ["/bin/sh", "-c", "while true; do sleep 3600; done"]
    volumeMounts:
    - mountPath: "/mnt/storage"
      name: my-storage
```