Deployment and Storage

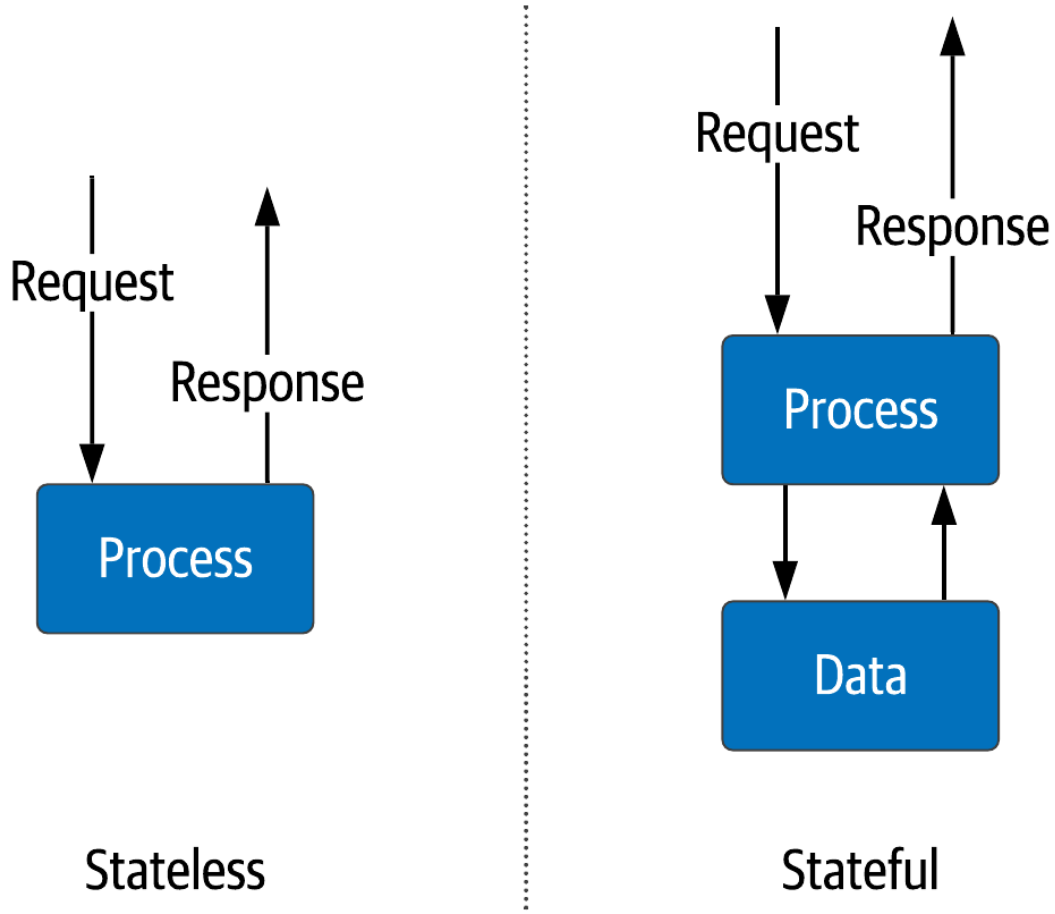
SoC Summer Workshop
Cloud Computing with Big Data

Richard T. B. Ma
School of Computing
National University of Singapore

Roadmap

- Stateless Applications
  - ReplicaSet
  - Deployment
- Stateful Applications
  - Storage: Volume, PersistentVolume (PV)
  - Configuration: EnvVar, ConfigMap

Stateless vs Stateful Applications



ReplicaSet (RS)

- Scenario: Pods may die, how to make a sustainable service?
- Solution: ReplicaSets represent a single, scalable microservice.
  - characteristic: every Pod created by RS controller is entirely homogenous.
- Kind-specific fields:
  - `replicas`: a fixed number of replicas
  - `template`: the pod definition template
  - `selector:matchLabels`: labels are used to filter Pod listings and track Pods running within a cluster

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: server-replicaset
spec:
  replicas: 2
  selector:
    matchLabels:
      app: server
  template:
    metadata:
      labels:
        app: server
    spec:
      containers:
        - name: server-container
          image: yancanmao/server-image
```

ReplicaSet -- matchExpressions

- matchExpressions option:
  - key: label key the selector applies to
  - operator: the relationship between the key & values, can be
    - In: the key must have a value in the specified list
    - NotIn: the key must not have a value in the specified list
    - Exists: the key must be present, regardless of its value
    - DoesNotExist: key must not be present
  - values: an array of string values, used with In and NotIn operators to specify acceptable values for the key

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
spec:
  replicas: 3
  selector:
    matchExpressions:
    - key: tier
      operator: In
      values:
      - frontend
    - key: environment
      operator: NotIn
      values:
      - dev
    - key: app
      operator: Exists
  template:
    metadata:
      labels:
        tier: frontend
        environment: production
        app: guestbook
    spec:
      containers:
      - name: php-redis
        image:
gcr.io/google_samples/gb-frontend:v3
        ports:
        - containerPort: 80
```

ReplicaSet – how does it work?

- RS guarantees the availability of a specified number of identical Pods and can scale via kubectl:

  $ kubectl scale replicasets server-replicaset --replicas=4

- Enabled by the reconcile or control loop of the RS controller that enables the application logic
  - observe # running pods `Replicas`
  - check the Diff with `Replicas`
  - increase or decrease Pod when Diff is not zero

- Inside Kubernetes Controller (pp. 16-30)
  - https://speakerdeck.com/govargo/inside-of-kubernetes-controller

Deployment

- Pod and RS are tied to fixed container images.
- Scenario: What if I want to upgrade my app in the image?
- Solution: Deployment specifies a preferred update strategy.
- Kind-specific field: `Strategy`
  - two `Types` of update strategies:
  1. `Recreate`
  2. `RollingUpdate`

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: server-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: server
  template:
    metadata:
      labels:
        app: server
    spec:
      containers:
      - name: server-container
        image: yancanmao/server-image
```

Deployment – typical use cases

- Scale up the Deployment to facilitate more load like RS.
  - To rollout a ReplicaSet. The RS creates Pods in the background.
- Manage the release of new versions
  - declare the new state of the Pods by updating the `PodTemplateSpec`
  - create a new RS and move Pods to the new one at a controlled rate
  - each new RS updates the revision of the Deployment.
- Rollback to an earlier revision.
  - Each rollback updates the revision of the Deployment.
- Pause the rollout to apply multiple fixes to its `PodTemplateSpec` and then resume it to start a new rollout.
- Use the status as an indicator that a rollout has stuck.

Deployment – how does it work?

- Check the status of the rollout

$ kubectl rollout status deployments <deployment-name>

- View the rollout history

$ kubectl rollout history deployment <deployment-name>

- Roll back to a previous rollout

$ kubectl rollout undo deployments <deployment-name>

- Pause a rollout

$ kubectl rollout pause deployments <deployment-name>

- Resume a paused rollout.

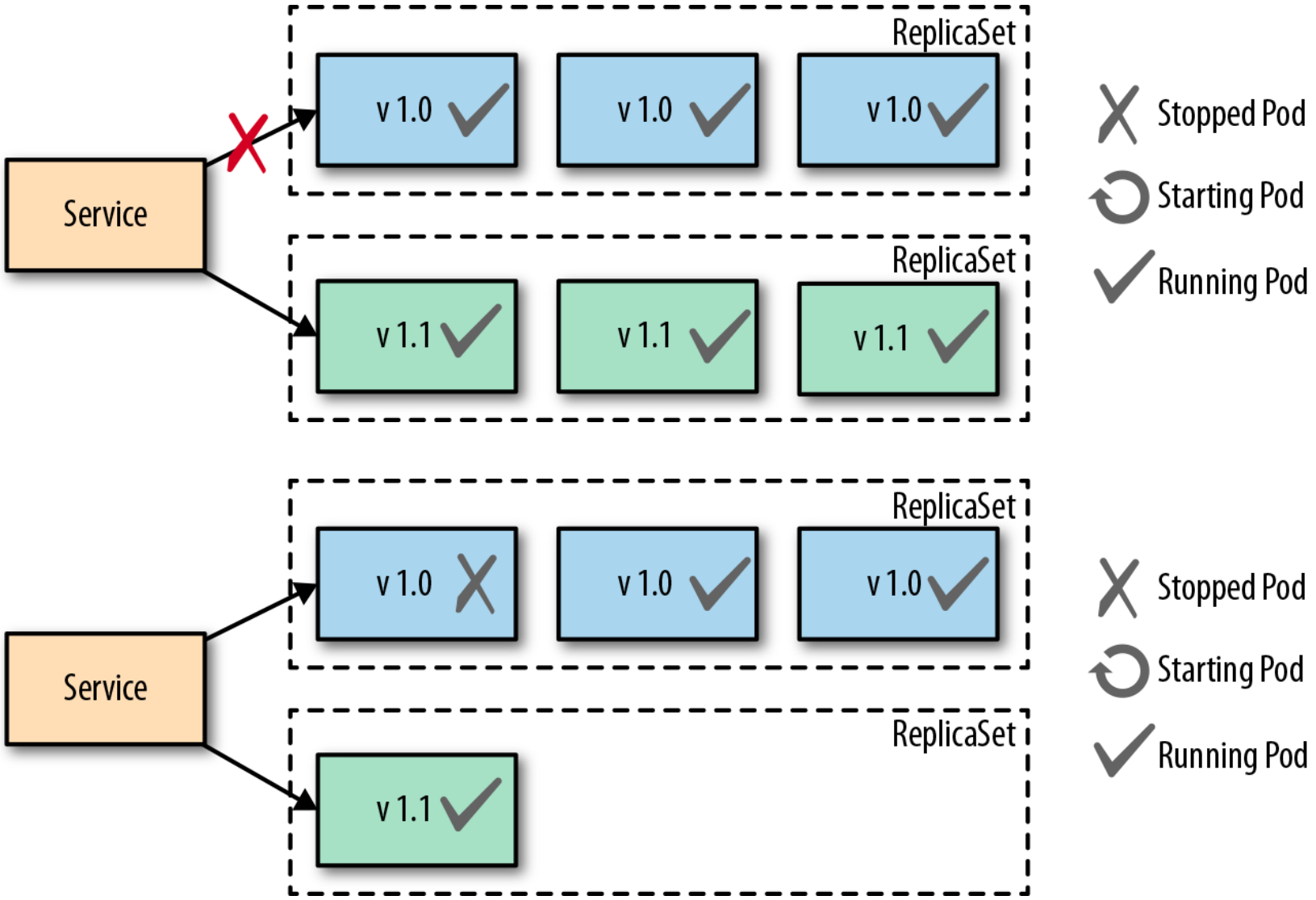
$ kubectl rollout resume deployments <deployment-name>

RollingUpdate Strategy

- When updating to a new version, a recreate strategy
  - terminates the old RS
  - starts pods of new image
- RollingUpdate uses two parameters:
  - maxUnavailable: the maximum number of Pods that can be unavailable during an update.
  - maxSurge: controls how many extra resources can be created to achieve a rollout.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rollingupdate-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: server
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: server
    spec:
      containers:
      - name: server-container
        image: yancanmao/server-image
```

Blue-Green (up) vs Canary (down) Release



Roadmap

- Stateless Application Deployment
  - ReplicaSet
  - Deployment
- Stateful Application Deployment
  - Storage: Volume, PersistentVolume, PVC
  - Configuration: EnvVar, ConfigMap

Storage outside K8s

- Where to save state/data?
  - external DB: `ExternalName` Service

```yaml
kind: Service
apiVersion: v1
metadata:
  name: external-database
spec:
  type: ExternalName
  externalName: my-db.com
```

- What if I don't have a URL for database, just an IP address?
  - create a Service with default type without a label selector
  - create an additional `Endpoints` object manually

```yaml
kind: Service
apiVersion: v1
metadata:
  name: external-ip-database
```

```yaml
kind: Endpoints
apiVersion: v1
metadata:
  name: external-ip-database
subsets:
  - addresses:
    - ip: 192.168.0.1
    ports:
    - port: 3306
```

Endpoints

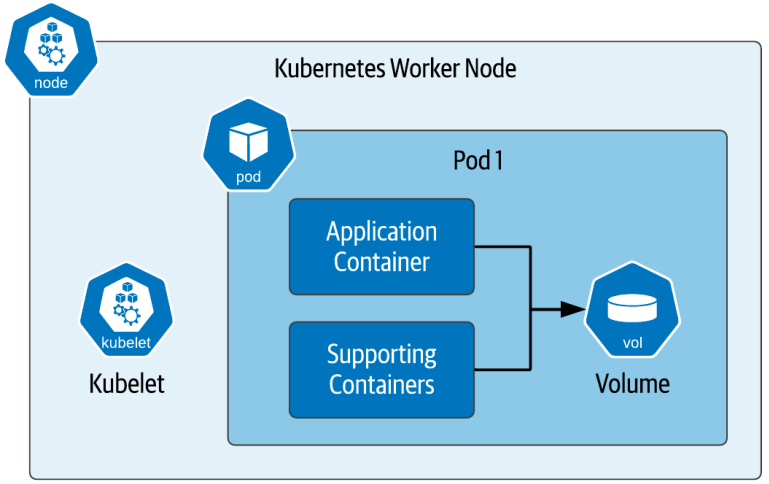
- Represents the network addresses and ports of Pods backing a Kubernetes Service
  - a K8s Service selects a set of Pods based on a label selector and groups them into an abstract endpoint
  - no spec field, which is commonly used for desired state
- Typical use cases
  - **Service Discovery and Load Balancing**: When serving a request to a Service, Endpoints determine available Pods and help distribute traffic.
  - **Pod Lifecycle Management**: Endpoints are dynamically updated based on changes to the underlying Pods. If Pods are added, removed, or updated (e.g., due to scaling, rolling updates, or failures), K8s updates them accordingly to reflect the current state of the available Pods.
  - **Network Policy Enforcement**: Endpoints help enforce Network Policies that can control access to Pods based on their labels and selectors.

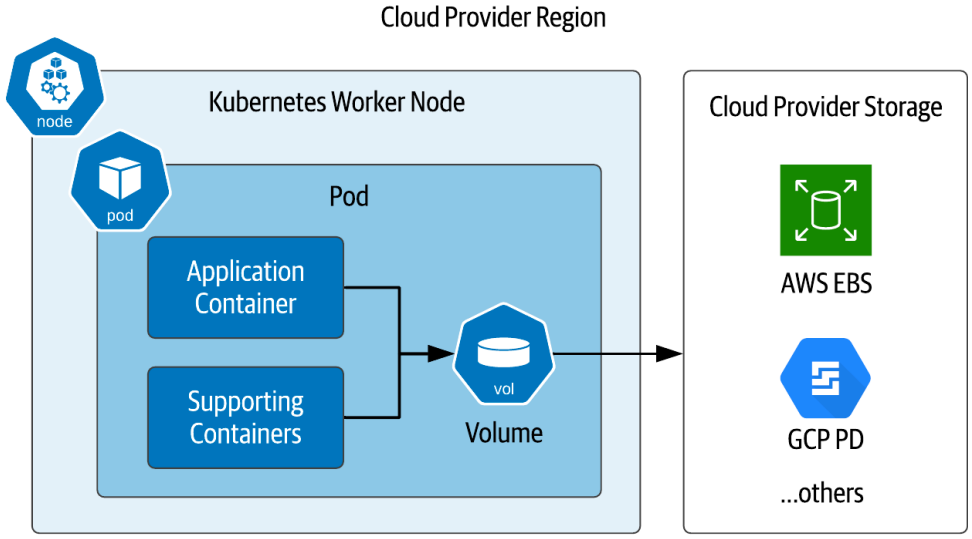
Storage inside K8s

- Limitation of external services: they do not perform any health checking.
  - the user is responsible for ensuring that the endpoint or DNS name is reliable for the apps
- Solution: let K8s cluster manage the data
  - ephemeral data
  - persistent data

Volume

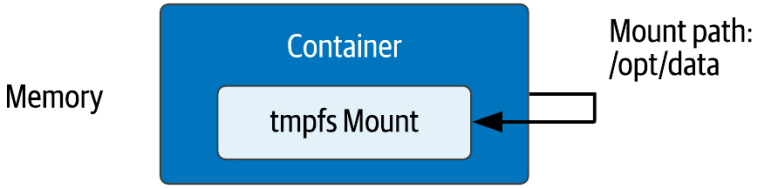
- An abstraction of storage that resides
  - on the nodes
  - remotely from third-party cloud providers






```yaml
apiVersion: v1
kind: Pod
metadata:
  name: emptydir-pod
spec:
  containers:
  - name: http-server
    image: yancanmao/app
    volumeMounts:
    - mountPath: /data
      name: temp
  volumes:
  - name: temp
    emptyDir: {}
```

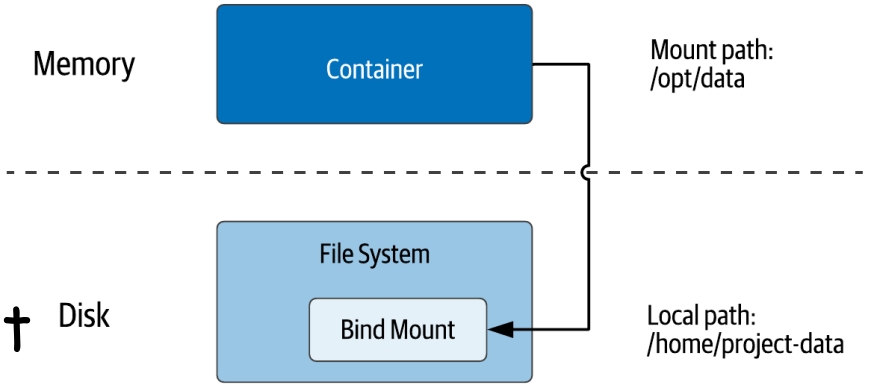
emptyDir Volume



- Ephemeral storage that starts out empty & provides a temp directory
- It persists only as long as the Pod is running on its node.
  - if the pod is removed from the node, the data in emptyDir is deleted permanently.
  - data stored on the node, either in memory or disk.
- Shared volume: ideal for sharing files between containers in a Pod
  - e.g., caching downloaded files or generated content, or using a scratch workspace for data processing jobs

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: emptydir-pod
spec:
  containers:
  - name: http-server
    image: yancanmao/app
    volumeMounts:
    - mountPath: /data
      name: temp
  volumes:
  - name: temp
    emptyDir:
      medium: Memory
```

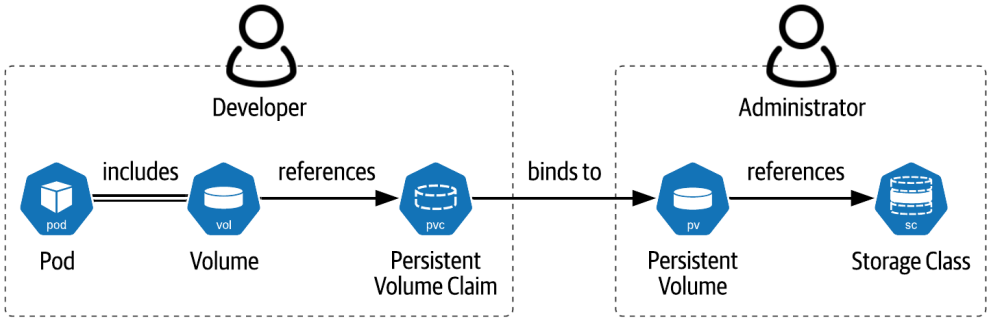
hostPath Volume



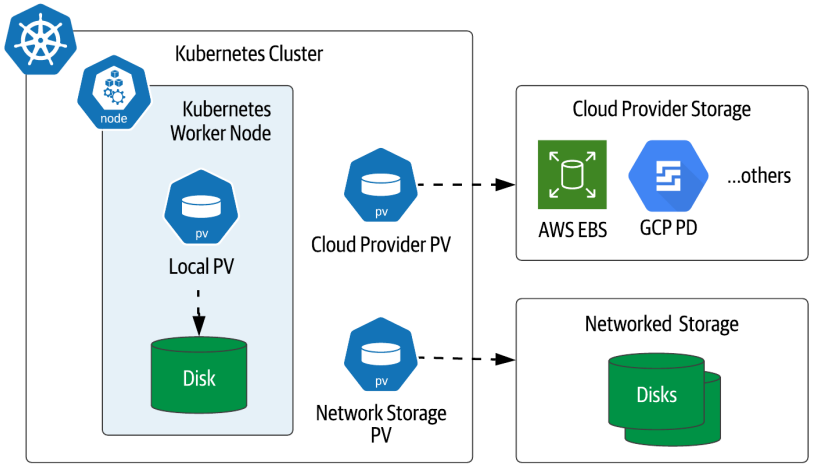
- Allow to mount from the host node's filesystem into a pod
- Mount type:
  - `EmptyDir`: an empty directory or create an empty directory if it doesn't exist.
  - `Directory`: a directory must exist.
  - `DirectoryOrCreate`: an existing directory or will be created if it does not exist.
  - `File`: a file that must exist on the host.
  - `FileOrCreate`: a file that exists or will be created if it does not exist.
  - `Socket`, `CharDevice`, `BlockDevice`: a UNIX socket, character device, a block device must exist on the host, respectively.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: hostpath-pod
spec:
  containers:
  - name: server-container
    image: yancanmao/server-image
    volumeMounts:
    - mountPath: /data
      name: host-data
  volumes:
  - name: host-data
    hostPath:
      path: /mnt/data
      type: Directory
```

Persistent Volumes



- The PersistentVolume subsystem consists of
  - PersistentVolume (PV)
  - PersistentVolumeClaim (PVC)
  - StorageClass
- PV provides a piece of storage in the cluster that
  - has been provisioned by an administrator, or
  - dynamically provisioned using Storage Classes
- Types of PVs
  - local
  - cloud provider
  - network storage



Persistent Volumes (PVs) – how to use?

- Step 1: cluster admin creates PVs
- Possible access modes:
  - ReadWriteOnce
  - ReadOnlyMany
  - ReadWriteMany
- Possible reclaim policies:
  - Retain, Recycle, Delete
- local PV type needs to specify nodeAffinity
  - unlike hostPath volume, restarted pods are rescheduled to the same node

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  local:
    path: /mnt/disks/ssd1
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - <node-name>
```

Persistent Volumes (PVs) – how to use?

- Step 2: app developer makes PVCs
- Step 3: pod mounts to the PVCs

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: local-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: local-pv-pod
spec:
  containers:
  - name: server-container
    image: server-image
    volumeMounts:
    - mountPath: "/data"
      name: local-storage
  volumes:
  - name: local-storage
    persistentVolumeClaim:
      claimName: local-pvc
```

How to Configure Applications?

- Scenario: Apps may need some configuration for accessing data sources, external services.
  - hardcode is bad, how can we externalize configs?
- One generic solution: using environment variables

❖ Dockerfile example:

```
FROM openjdk:11
ENV SEED "1349093094”
ENV LOG_FILE "/tmp/random.log"

# Alternatively:
LOG_FILE=/tmp/random.log
SEED=1349093094
```

❖ Read env variables in Java:

```
Long seed =
Long.parseLong(System.getenv("
SEED"));
```

❖ Set/change env variables:

```
docker run -e
LOG_FILE="/tmp/random.log" \
-e SEED="147110834325" \
yancanmao/server-image
```

EnvVar in K8s

- For a handful simple configurations
- Limitations:
  - difficult to find/retrieve
  - env variables defined within a Docker image can be replaced during runtime in Deployment
  - can be set only before an application starts, and we cannot change them later

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: server-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: server
  template:
    metadata:
      labels:
        app: server
    spec:
      containers:
      - name: server-container
        image: yancanmao/server-image
        env:
        - name: LOG_FILE
          value: /tmp/random.log
        - name: SEED
          value: 1349093094
```

ConfigMap (CM)

- A config file that storages & manages of key-value pairs
  - key as env variable name
  - key as filename

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: server-configmap
data:
  PATTERN: Configuration Resource
  application.properties: |
    # Random Generator config
    log.file=/tmp/generator.log
    server.port=7070
  SEED: "432576345"
```

- Create ConfigMap using kubectl

```
$ kubectl create cm server-configmap \
--from-literal=PATTERN="Configuration Resource" \
--from-literal=SEED="432576345" \
--from-file=application.properties
```

ConfigMap - how to use?

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: server-pod
spec:
  containers:
    - env:
      - name: PATTERN
        valueFrom:
          configMapKeyRef:
            name: server-configmap
            key: PATTERN
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: server-pod
spec:
  containers:
    envFrom:
    - configMapRef:
        name: server-configmap
      prefix: CONFIG_
```

- Specify environment variable
  - in Pod specification
  - in Pod template in Depolyment & ReplicaSet
- env field with parameter configMapKeyRef:
  - name of ConfigMap
  - key of env var to import
- envFrom field with configMapRef parameter:
  - prefix used to import all vars whose keys start with it

ConfigMap – how to use?

```yaml
apiVersion: v1
kind: Pod
metadata:
    name: configmap-pod
spec:
  containers:
  - image: yancanmao/server-image
    name: server-container
    volumeMounts:
    - name: config-volume
        mountPath: /config
  volumes:
  - name: config-volume
    configMap:
        name: server-configmap
```

- CM-backed volume will contain as many files as entries
  - the map's keys as filenames
  - the map's values as file content
- Files in a mounted CM volume is updated when the ConfigMap is updated via K8s API.
  - can support apps with hot reload of configurations
  - however, if used as env variables, updates are not reflected, as env variables can't be changed after a process has been started.