Kubernetes: Persistent Storage

Contacts

Péter Megyesi

Dávid Szabó

megyesi@leannet.eu
twitter.com/M3gy0
linkedin.com/in/M3gy0

szabo@leannet.eu
twitter.com/szabvid
linkedin.com/in/szabvid

Course Outline

1. What is Kubernetes?
   - Components
   - Installation
2. Basics of Docker
   - Namespaces
   - Building and running Docker images
3. Pods and Deployments
   - Running basic workloads in Kubernetes
   - Scale, Update, Rollback
4. Advanced Pod configuration
   - Args, Envs, ConfigMaps, Secrets
   - Init- and sidecar containers
   - Scheduling and debugging
5. Networking in Kubernetes
   - What are network plugins?
   - Service abstraction and ingress
6. **Persistent storage**
   - **Basics of storage: block vs. object vs. file system**
   - **StoragesClass, PVC, PV**
7. Security
   - RBAC: Roles, ServiceAccounts, RoleBindings
   - Security context and network security policy
8. Advanced topics
   - Helm
   - Custom resources and operators

Storage is a Complicated Problem

Kelsey Hightower @kelseyhightower · Mar 24

Some people believe that rubbing Kubernetes on a stateful workload turns it into a fully managed database offering rivaling RDS. This is false. Maybe with enough effort, and additional components, and an SRE team, you can build RDS on top of Kubernetes.

20 153 492

Show this thread

f in

Why Is Storage On Kubernetes So Hard?

By **Gokhan Simsek**

Article | Friday, January 11 2019

Tim Hockin @thockin · 26 Oct 2018

Replying to @stu

Storage is **hard**. Maybe the hardest of all the infra problems. It's not surprising that it is lagging. The need for really dynamic cluster FSes has never been greater, but those things take a long time to stabilize. #kubernetes

[LIVE CHAT] Kubernetes Influencer Chat

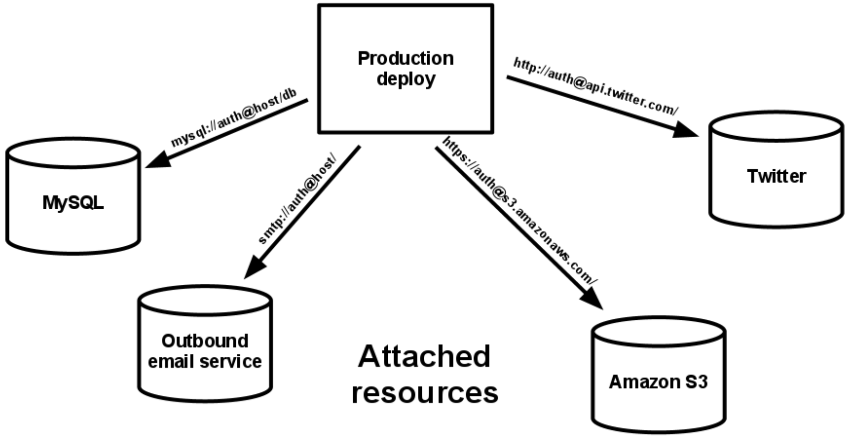
Last year at #kubecon there was a lot of tension around storage - Rook, OpenEBS, etc. It's been the lagging piece of the stack, has progress been made?

crowdchat.net

1

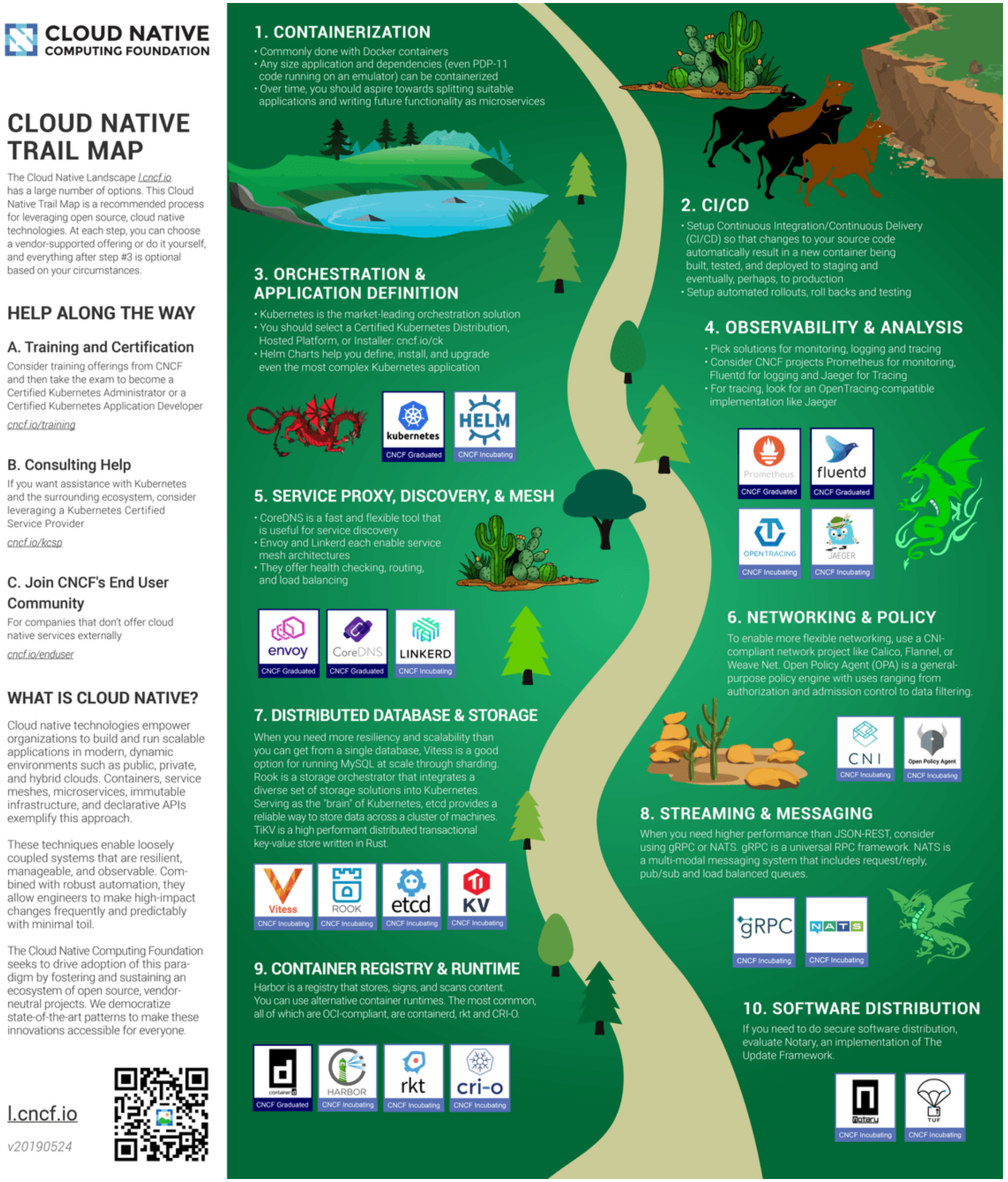
Remember: 12 Factor App – IV: Backing Services

- Treat backing services as attached resources -> loose coupling
- *Backing service* is any service the app consumes over the network as part of its normal operation, e.g.:
  - datastores (MySQL)
  - messaging/queueing systems (RabbitMQ)
  - metrics-gathering services (New Relic)
  - and even API-accessible consumer services (Twitter, Google Maps)



- Services should be easily interchangeable: referencing them as simple URLs with login credentials
- This will ensure good portability and helps maintain your system:
  - E.g. swap out a local MySQL database with Amazon RDS without any changes to the app’s code

Also Remember: Cloud Native Trail Map – DB and Storage is Only Step 7



CLOUD NATIVE COMPUTING FOUNDATION

CLOUD NATIVE TRAIL MAP

The Cloud Native Landscape *l.cncf.io* has a large number of options. This Cloud Native Trail Map is a recommended process for leveraging open source, cloud native technologies. At each step, you can choose a vendor-supported offering or do it yourself, and everything after step #3 is optional based on your circumstances.

HELP ALONG THE WAY

A. Training and Certification

Consider training offerings from CNCF and then take the exam to become a Certified Kubernetes Administrator or a Certified Kubernetes Application Developer

cncf.io/training

B. Consulting Help

If you want assistance with Kubernetes and the surrounding ecosystem, consider leveraging a Kubernetes Certified Service Provider

cncf.io/kcsp

C. Join CNCF's End User Community

For companies that don't offer cloud native services externally

cncf.io/enduser

WHAT IS CLOUD NATIVE?

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

The Cloud Native Computing Foundation seeks to drive adoption of this paradigm by fostering and sustaining an ecosystem of open source, vendor-neutral projects. We democratize state-of-the-art patterns to make these innovations accessible for everyone.

l.cncf.io

v20190524

1. CONTAINERIZATION

- Commonly done with Docker containers
- Any size application and dependencies (even PDP-11 code running on an emulator) can be containerized
- Over time, you should aspire towards splitting suitable applications and writing future functionality as microservices

2. CI/CD

- Setup Continuous Integration/Continuous Delivery (CI/CD) so that changes to your source code automatically result in a new container being built, tested, and deployed to staging and eventually, perhaps, to production
- Setup automated rollouts, roll backs and testing

3. ORCHESTRATION & APPLICATION DEFINITION

- Kubernetes is the market-leading orchestration solution
- You should select a Certified Kubernetes Distribution, Hosted Platform, or Installer: cncf.io/ck
- Helm Charts help you define, install, and upgrade even the most complex Kubernetes application

kubernetes — CNCF Graduated
HELM — CNCF Incubating

4. OBSERVABILITY & ANALYSIS

- Pick solutions for monitoring, logging and tracing
- Consider CNCF projects Prometheus for monitoring, Fluentd for logging and Jaeger for Tracing
- For tracing, look for an OpenTracing-compatible implementation like Jaeger

Prometheus — CNCF Graduated
fluentd — CNCF Graduated
OPENTRACING — CNCF Incubating
JAEGER — CNCF Incubating

5. SERVICE PROXY, DISCOVERY, & MESH

- CoreDNS is a fast and flexible tool that is useful for service discovery
- Envoy and Linkerd each enable service mesh architectures
- They offer health checking, routing, and load balancing

envoy — CNCF Graduated
CoreDNS — CNCF Graduated
LINKERD — CNCF Incubating

6. NETWORKING & POLICY

To enable more flexible networking, use a CNI-compliant network project like Calico, Flannel, or Weave Net. Open Policy Agent (OPA) is a general-purpose policy engine with uses ranging from authorization and admission control to data filtering.

CNI — CNCF Incubating
Open Policy Agent — CNCF Incubating

7. DISTRIBUTED DATABASE & STORAGE

When you need more resiliency and scalability than you can get from a single database, Vitess is a good option for running MySQL at scale through sharding. Rook is a storage orchestrator that integrates a diverse set of storage solutions into Kubernetes. Serving as the "brain" of Kubernetes, etcd provides a reliable way to store data across a cluster of machines. TiKV is a high performant distributed transactional key-value store written in Rust.

Vitess — CNCF Incubating
ROOK — CNCF Incubating
etcd — CNCF Incubating
KV — CNCF Incubating

8. STREAMING & MESSAGING

When you need higher performance than JSON-REST, consider using gRPC or NATS. gRPC is a universal RPC framework. NATS is a multi-modal messaging system that includes request/reply, pub/sub and load balanced queues.

gRPC — CNCF Incubating
NATS — CNCF Incubating

9. CONTAINER REGISTRY & RUNTIME

Harbor is a registry that stores, signs, and scans content. You can use alternative container runtimes. The most common, all of which are OCI-compliant, are containerd, rkt and CRI-O.

containerd — CNCF Graduated
HARBOR — CNCF Incubating
rkt — CNCF Incubating
cri-o — CNCF Incubating

10. SOFTWARE DISTRIBUTION

If you need to do secure software distribution, evaluate Notary, an implementation of The Update Framework.

Notary — CNCF Incubating
TUF — CNCF Incubating

What Storage Type an Application Needs

Raw storage vs. Database

- Block Storage
  - iSCSI
  - Fibre Channel
  - GCE Persistent Disks
  - Amazon EBS
  - Local Disks
- File Storage
  - NFS
  - SMB
  - GlusterFS
  - CephFS

- SQL Databases
  - MySQL
  - PostgreSQL
  - SQL Server
- NoSQL Databases
  - Key-value or document based
  - MongoDB
  - Redis
  - Cassandra
- Time series Databases
  - InfluxDB
  - Prometheus
  - Graphite

- Message Queues
  - Apache Kafka
  - RabbitMQ
  - NATS
  - Apache Camel
  - Google Cloud Pub/Sub
  - Amazon SQS
- Object Stores
  - Amazon S3
  - Google Cloud Storage (GCS)
  - Azure Blob Storage
  - MinIO

What Storage Type an Application Needs

- What does your stateful app need?
- What kind of data are you storing?
- Where will you be accessing it?
- How frequently will you need to access it?
- What kind of data protection is required?

Availability

Durability

Performance

Cost

Managed vs. Unmanaged

Stateful App	**Your stateful app**
Data Service	**Object Store, SQL/NoSQL DB, Message Queue, etc.**
Block/File Storage	**NFS, iSCSI Fibre Channel, etc.**
Physical Storage	**HDD, SSD, NVMe, Flash Disk**

Databases as Kubernetes pods

You can decide whether these layers should be managed by Kubernetes, our should be external

Software Defined Storage as Kubernetes pods

Decide in which layer you want to provide High Availability (HA) for you application data
→ Data replication

The Key Takeaway

Stateful applications, especially database clusters and SDS tools have very complicated life cycles

Use operators to manage such stateful apps inside Kubernetes

Stateless vs. Stateful Workflow in Kubernetes

- Stateless apps:

  if Pod (or container inside) crashes --> storage and file abstractions vanish

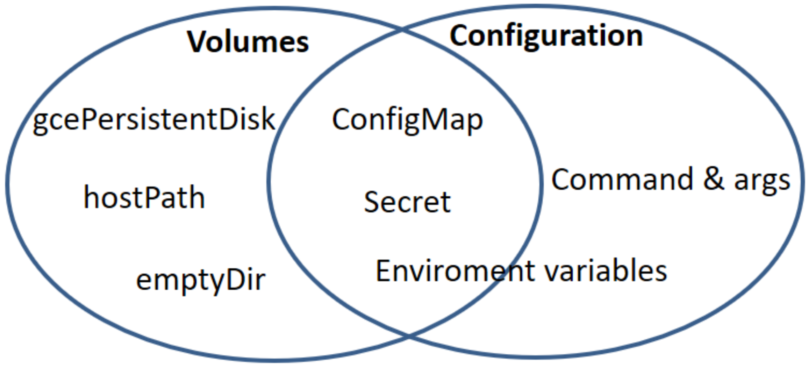
- Stateful apps:

  attached volumes help to persist data

In K8s a volume serves two purpose:

1. Persistence:
   - storage that lasts longer than the container **or**
   - lasts longer than the Pod

   depends on the volume type
2. Shared state/configuration:
   - can be used by multiple containers in a Pod to share state/configuration



Assigning Volumes to Containers

It is a 3 step logical process:

1. Creating the volume → A developer or cloud admin responsibility
2. Defining the volume in the Pod yaml
3. Mounting the volume into the Container(s)

It's definitely a developer responsibility

Take a look into creating the volume:

– The need of manually creating a volume depends on the volume type:
  - emptyDir --> created automatically and exists until the Pod lives
  - hostPath --> uses a folder on the file system (which can be a native folder or backed by an NFS, Gluster, Ceph, etc.)

– It would be nice to decouple the **creation** and the **usage** of the volume:
  - Creation: the admin/operator creates a PersistentVolume object that abstract away the storage from the developer
  - Developer: uses a PersistentVolumeClaim without the need of creating or knowing much about the storage behind
    – So at the end of the day the PersistentVolumeClaim is just another volume type that can be used
    – It is assaigns a pre-provisioned PersistentVolume to the Pod via the PersistentVolumeClaim

Volume Types

1. Not for storage but for configuration:
   – configMap, secret, downwardAPI, emptyDir, projected
2. Storage but not abstracts away the storage details (hence make volume usage “coupled”):
   – awsElasticBlockStore, azureDisk, azureFile, cephfs, cinder, csi, fc (fibrechannel), flexVolume, flocker, gcePersistentDisk, glusterfs, hostPath, iscsi, local, nfs, portworxVolume, quobyte, rbd, scaleIO, storageOS, vsphereVolume
3. Storage that abstracts away the storage details:
   – PersistentVolumeClaim: refers to a PersistentVolume

```
volumes:
- name: test-volume
  # GCE PD must already exist.
  gcePersistentDisk:
    pdName: my-data-disk
    fsType: ext4
```

```
volumes:
- name: mypd
  persistentVolumeClaim:
    # PVC reference
    claimName: myclaim
```

https://kubernetes.io/docs/concepts/storage/volumes/#types-of-volumes

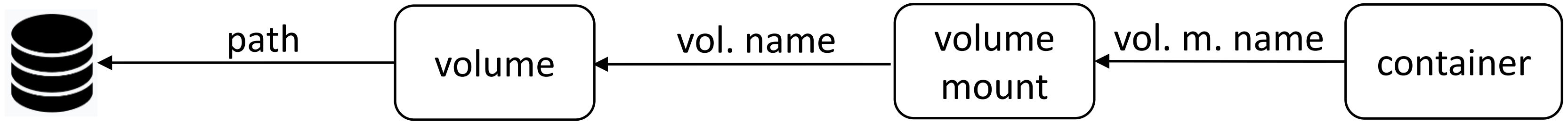
Difference between non-PVC and PVC volumes

Without PVC:

1 step:

```
volumes:
  - name: config-vol
    hostPath:
      path: /data
```

```
volumeMounts:
  - name: config-vol
    mountPath: /etc/config
```



Developer takes **care** of **everything** when he creates the Pod definition

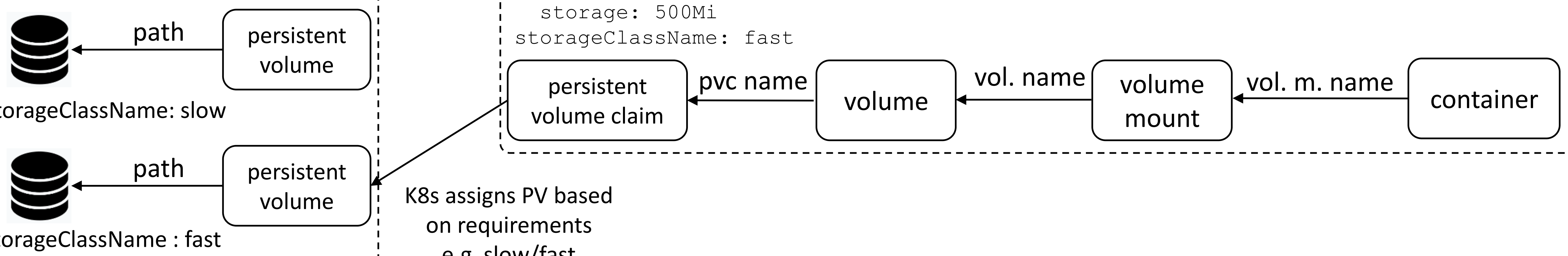
With PVC:

1. step: Operator creates persistent volumes

2. step:

Developer takes **care** only to define app **storage requirements**

```
requests:
  storage: 500Mi
storageClassName: fast
```



K8s assigns PV based on requirements e.g. slow/fast

Realistic PV and PVC definitions

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  nfs:
    path: /tmp
    server: 172.17.0.2
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: slow
```

accessModes:

- Three options:
  1. ReadWriteOnce: can be mounted as read-write by a single node
  2. ReadOnlyMany: can be mounted read-only by many nodes
  3. ReadWriteMany: can be mounted as read-write by many nodes
- Depends on the resource provider:

Volume Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
AWSElasticBlockStore	✓	-	-
AzureFile	✓	✓	✓
AzureDisk	✓	-	-
CephFS	✓	✓	✓
Cinder	✓	-	-
CSI	depends on the driver	depends on the driver	depends on the driver
FC	✓	✓	-
FlexVolume	✓	✓	depends on the driver
Flocker	✓	-	-
GCEPersistentDisk	✓	✓	-

Realistic PV and PVC definitions

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  nfs:
    path: /tmp
    server: 172.17.0.2
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: slow
```

accessModes:

- Three options:
  1. ReadWriteOnce: can be mounted as read-write by a single node
  2. ReadOnlyMany: can be mounted read-only by many nodes
  3. ReadWriteMany: can be mounted as read-write by many nodes
- Depends on the resource provider:

Volume Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
Glusterfs	✓	✓	✓
HostPath	✓	-	-
iSCSI	✓	✓	-
Quobyte	✓	✓	✓
NFS	✓	✓	✓
RBD	✓	✓	-
VsphereVolume	✓	-	- (works when Pods are collocated)
PortworxVolume	✓	-	✓
ScaleIO	✓	✓	-
StorageOS	✓	-	-

Realistic PV and PVC definitions

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  nfs:
    path: /tmp
    server: 172.17.0.2
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: slow
```

storageClassName:

- A PV can only be bound to PVCs requesting that class.
- A PV with no storageClassName has no class and can only be bound to PVCs that request no particular class.

reclaimPolicy:

- What should happen after a PVC is deleted?
- Retain:
  - manual reclamation
- Recycle:
  - basic scrub (rm -rf /thevolume/*)
  - deprecated (dynamic provisioning should be used)
- Delete:
  - associated storage asset such as AWS EBS, GCE PD, Azure Disk, or OpenStack Cinder volume is deleted

volumeMode:

- Filesystem: mounted into Pods into a directory
- Block: use a volume as a raw block device
  - app needs to know how to handle it

Realistic PV and PVC definitions

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  nfs:
    path: /tmp
    server: 172.17.0.2
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: slow
```

Rules of binding PVC to PV:

- One-to-one relationship:
  - Exactly one PVC can be bound to a PV
  - Even if the PV is much larger that the request in the PVC
  - K8s tries to assign optimally based on:
    - Sufficient capacity
    - Access mode
    - Volume mode
    - Storage class
  - If developer assumes multiple possibilities can use selectors in PVC
- If PVC cannot be bound then it remains in `PENDING` state:
  - until a proper PV is not created
- Storage Object in Use Protection:
  - No PV or PVC can be deleted until released
  - The order is:
    - Deleting Pod that uses PVC
    - Deleting PVC that uses PV
    - Deleting PV
- PVC and Pod need to be in the same namespace (PV has “cluster scope”)

Static vs. Dynamic Storage Provisioning

What we saw so far is **static provisioning**:

- The creation of PV decoupled from the creation of the Pod
  - 1. step: administration provision PVs
  - 2. step: developer defined PVCs
  - But still an administrator must do the 1. step manually

How **dynamic provisioning** is different:

- The creation of PV is automated, no administrator is needed
- A provisioner (e.g. Google Storage) can do that based on a StorageClass object
- In the StorageClass we define:
  - provisioner
  - type
  - replication-type
  - …

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  fstype: ext4
  replication-type: none
```

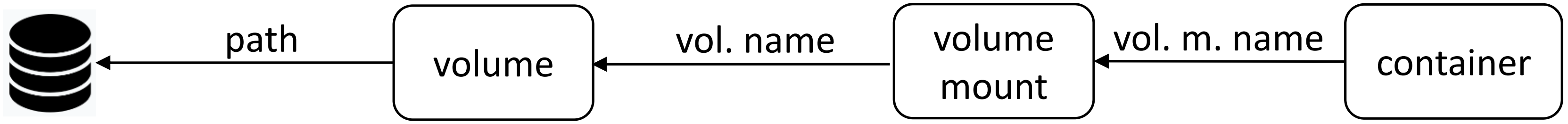
Static vs. Dynamic Storage Provisioning

Without PVC:

1 step:

```
volumes:
  - name: config-vol
    hostPath:
      path: /data
```

```
volumeMounts:
  - name: config-vol
    mountPath: /etc/config
```



Developer takes **care** of **everything** when he creates the Pod definition

With PVC (dynamic provision):

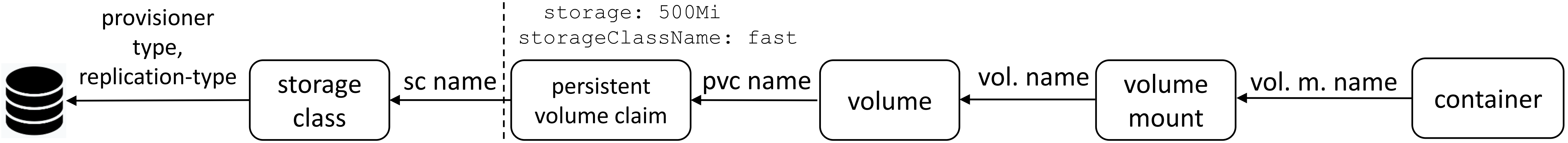
3. step:

1. step:

2. step:

Developer takes **care** only to define app **storage requirements**

```
requests:
  storage: 500Mi
storageClassName: fast
```



Dynamic Storage Provisioning with Rook

- Cloud-native storage orchestrator for Kubernetes
- Provides self-managing, self-scaling, and self-healing storage services
- Automates deployment, bootstrapping, configuration, provisioning, scaling, upgrading, migration, disaster recovery, monitoring, and resource management
- Provides a platform, framework, and support for a diverse set of storage solutions:
  - **Ceph (stable)**
  - EdgeFS (stable)
  - NFS, CockroachDB, Cassandra, YugabyteDB (alpha)
- CNCF Incubating project

Dynamic Storage Provisioning with OpenEBS

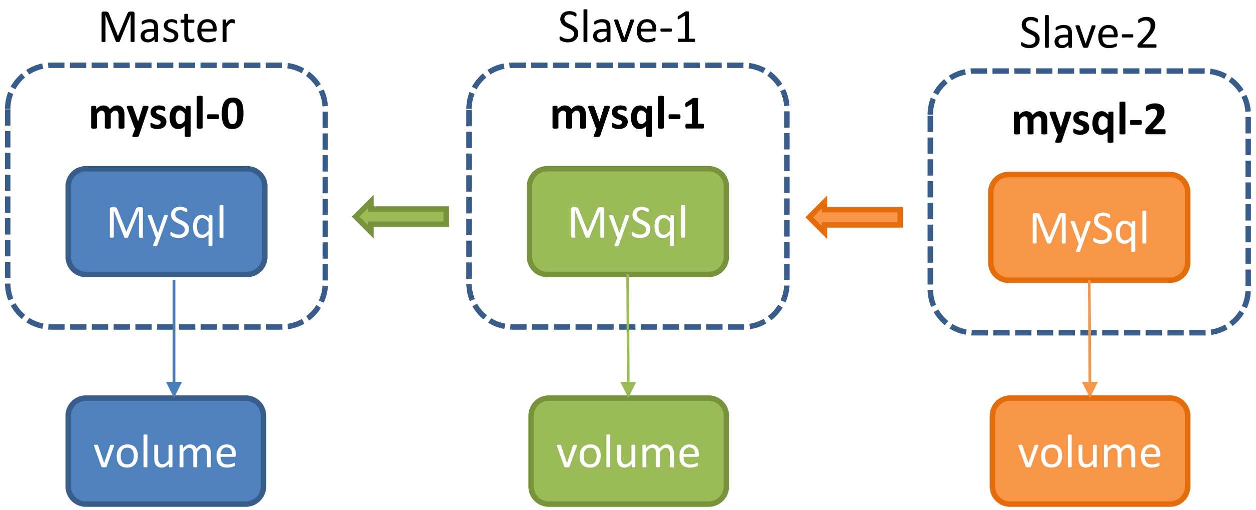
- Easy to use open-source storage solution for Kubernetes
- Built completely in userspace making it highly portable to run across any OS/platform
- Supports a range of storage engines so that developers can deploy the storage technology appropriate to their application design objectives
- Enables easy access Dynamic Local PVs or Replicated PVs
- CNCF Sandbox project

Dynamic Storage Provisioning with Longhorn

- Distributed block storage system for Kubernetes
- Lightweight, reliable, and powerful
- Dedicated storage controller for each block device volume and synchronously replicates the volume across multiple replicas stored on multiple nodes
- Main features:
  - Enterprise-grade distributed storage with no single point of failure
  - Incremental snapshot of block storage
  - Backup to secondary storage (NFSv4 or S3-compatible object storage) built on efficient change block detection
  - Recurring snapshot and backup
  - Automated non-disruptive upgrade
  - Intuitive GUI dashboard
- CNCF Incubating project

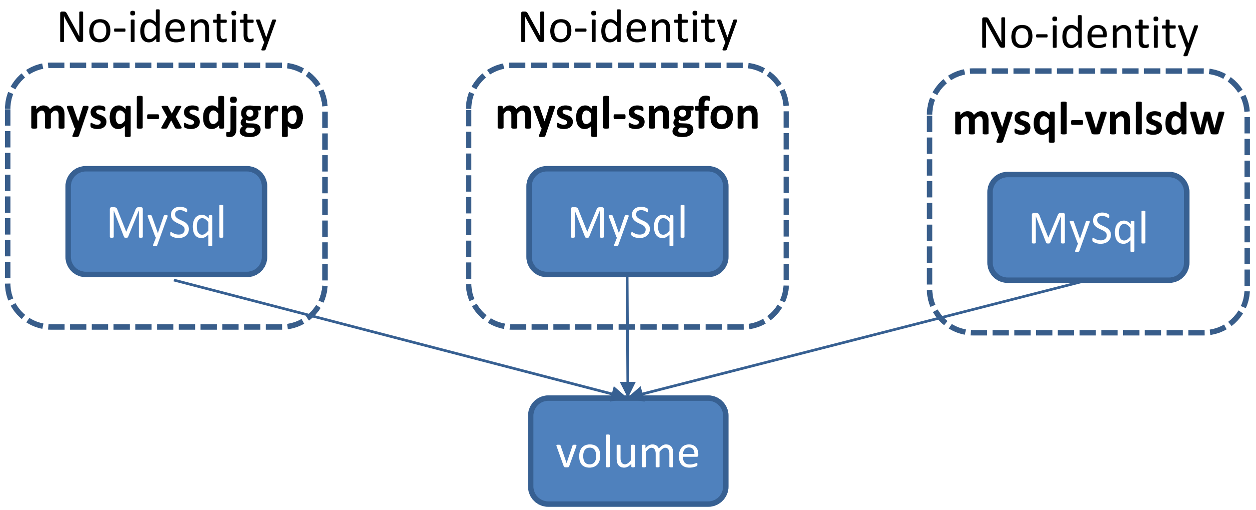
StatefulSet

- A Deployment alternative for managing stateful applications
- Why does deployment not enough?
- Let’s say we want a HA database deployment:
  - One master and two read replicas
  - The usual process is to:
    - Write to Master
    - Sync Replica 1 from Master
    - Sync Replica 2 from Replica 1



StatefulSet

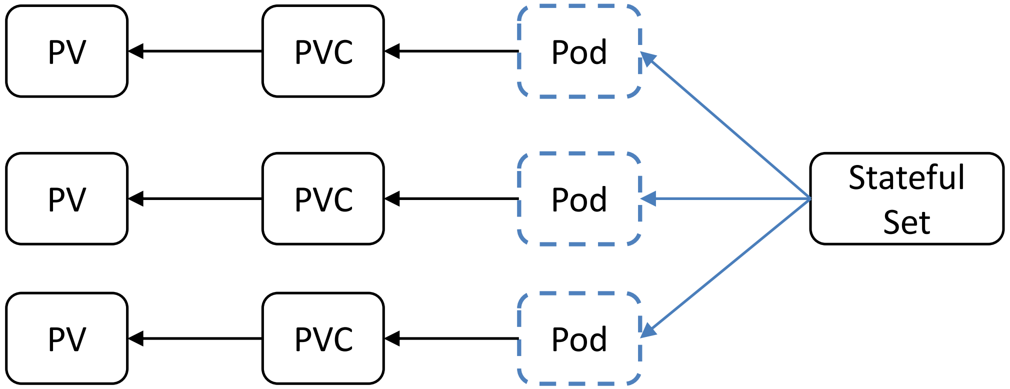
- A Deployment alternative for managing stateful applications
- Why does deployment not enough?
- Let’s say we want a HA database deployment:
  - One master and two read replicas
  - The usual process is to:
    - Write to Master
    - Sync Replica 1 from Master
    - Sync Replica 2 from Replica 1



- Problems:
  - Deployment cannot give Pods identity (Master, Replica-1, Replica-2)
  - Deployment cannot guarantee to bring up Pods in order (1: Master, 2: Replica-1, 3: Replica-2)
  - Deployment cannot assign PV to each Pod

StatefulSet

- Provides guarantees about:
  - Uniqueness of the Pods (by default)
  - Ordered, graceful deployment and scaling (by default)
  - Stable, persistent storage for each Pod:
    - By dynamic storage provisioning (storage classes)
    - By creating PVs manually, a priori
  - Stable, unique network identifiers (by Headless Service):
    - Headless Service can assign unique DNS entries to each Pod
    - serviceName should point to such Service
  - If Pod crashes it will be replaced and has the same identity as the replaced one.




```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  …
spec:
  selector:
    …
  serviceName: nginx
  replicas: 3
  template:
    metadata:
      …
    spec:
      containers:
      - name: nginx
        image: nginx
        volumeMounts:
          …
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "my-storage-class"
      resources:
        requests:
          storage: 1Gi
```

StatefulSet

- Considerations:
  - Deleting and/or scaling a StatefulSet down will *not* delete the volumes associated with the StatefulSet.
  - StatefulSets do not provide any guarantees on the termination of pods when a StatefulSet is deleted.
    - Instead scale it down to 0 prior to deletion
  - StatefulSets currently require a Headless Service to be responsible for the network identity of the Pods.
  - Storage for a given Pod must either be provisioned by a PersistentVolume Provisioner.

Horizontal Pod Autoscaler

- Automatically scales the number of Pods in a:
  - replication controller, deployment, replica set or stateful set
- Based on:
  - observed CPU utilization (by default)
  - custom metric provided by application
- Algorithm:
  - `desiredReplicas = ceil[currentReplicas * ( currentMetricValue / desiredMetricValue )]`
  - E.g.
    - If currentMetricValue = 200m and desiredMetricValue = 100m --> doubles the number of replicas
    - If currentMetricValue = 100m and desiredMetricValue = 200m --> halves the number of replicas
    - If the ratio is sufficiently close to 1.0 --> does not react
      - (can be controlled by: `--horizontal-pod-autoscaler-tolerance`)
- HPA cannot scale the cluster!
  - If nodes do not have enough resources for the extra pods, HPA cannot do anything about it.
  - Cluster scaling is usually an option for managed Kubernetes clusters

Horizontal Pod Autoscaler

- HPA cannot work without the Metrics Server:
  - It can be considered as a cluster add-on
  - https://github.com/kubernetes-sigs/metrics-server
  - For managed clusters it is usually installed automatically
  - For self-managed clusters an installation is necessary
  - Among others “kubectl” related commands also use the metrics server to get information:
    - `kubectl top pods`
    - `kubectl top nodes`
- Autoscaling on multiple metrics is possible:
  - CPU
  - Memory
  - Packets per second
  - Requests per second

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  maxReplicas: 10
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  targetCPUUtilizationPercentage: 50
status:
  currentReplicas: 0
  desiredReplicas: 0
```