Kubernetes

Anurag Abhay Sarathy
Sr. Software Engineer

VERITAS

Disclaimer

Several images, illustrations, and examples in this presentation are borrowed from the book ”Kubernetes in Action” by Manning Publications. Please limit the use of this presentation to only the audience of this lecture and please do not publish it publicly.

VERITAS

Concepts

Declarative vs Imperative Configuration

- Declarative Configuration
  - One of the primary drivers behind development of K8S.
  - Specify the desired state
  - K8S understands the desired state, it can take autonomous action, independent of user interaction
  - K8S can implement autonomous self-correcting and self-healing behaviors
  - Statement – “I want there to be five replicas of my web server running at all times.”
  - K8S ensures that the state above is reached.
- Imperative Configuration
  - User takes series of direct actions (e.g. “run this”)
  - Simpler to understand but needs user interaction

VERITAS

Concepts

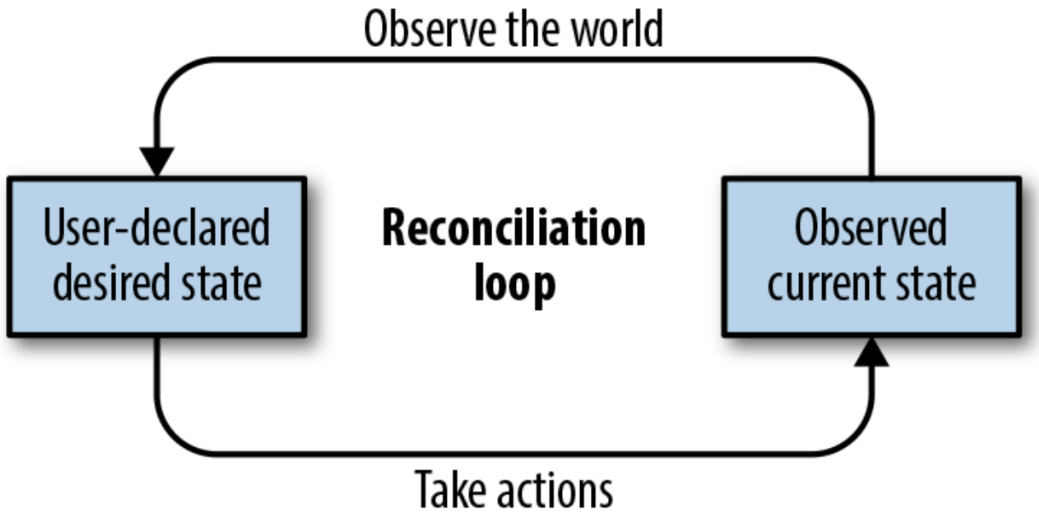
Reconciliation or Controllers

- Monolithic System Design
  - System is aware of the entire world and uses complete view to move everything forward in a coordinated fashion
  - Centralized system and easy to understand/debug
  - Not stable, a single issue can bring down the entire system
- Decentralized System Design
  - K8S is composed of large number of controllers
  - Every controller
    - Runs it's own independent reconciliation loop
    - Responsible for specific piece of the system
  - More stable system
  - Large number of independent processes interoperate to achieve the goal
  - Difficult to understand the system and debug

VERITAS

Concepts

- Control Loop
  1. Obtain the desired state of the world
  2. Observe the world
  3. Find difference between observation of the world and desired state of the world
  4. Take actions to make the observation of the world match the desired state

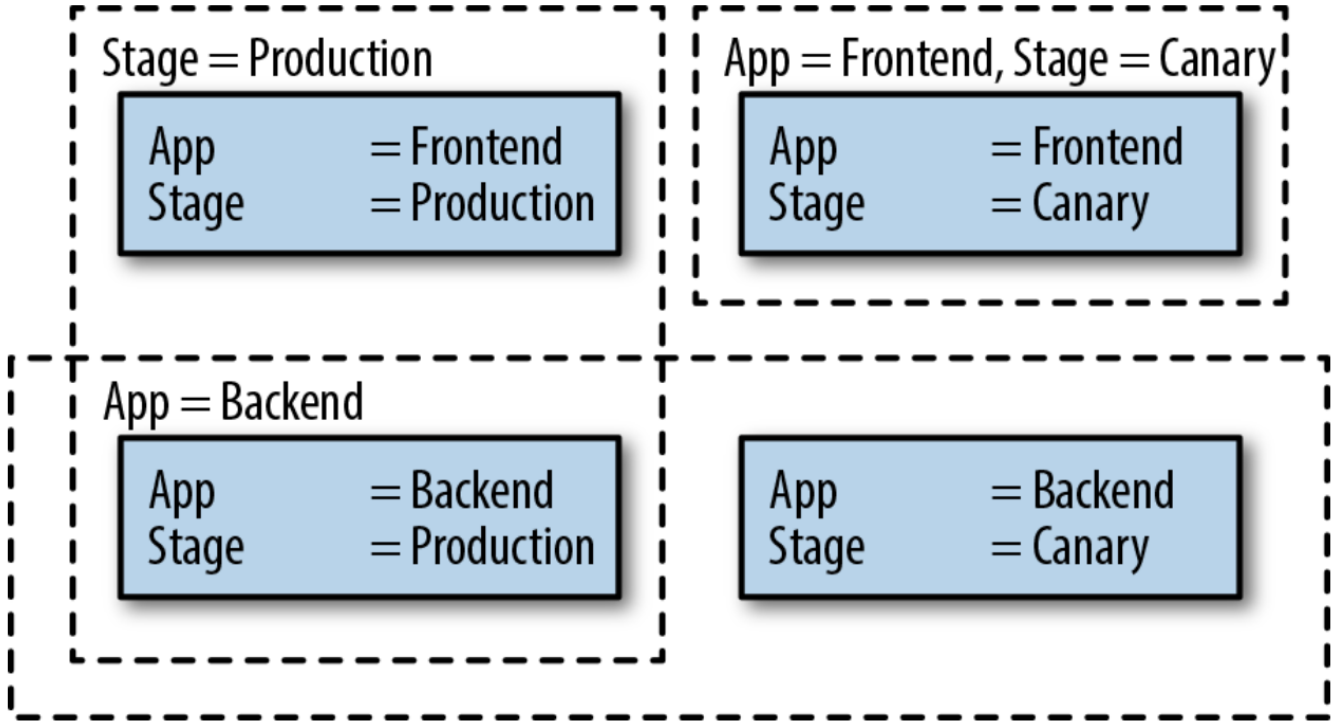


VERITAS

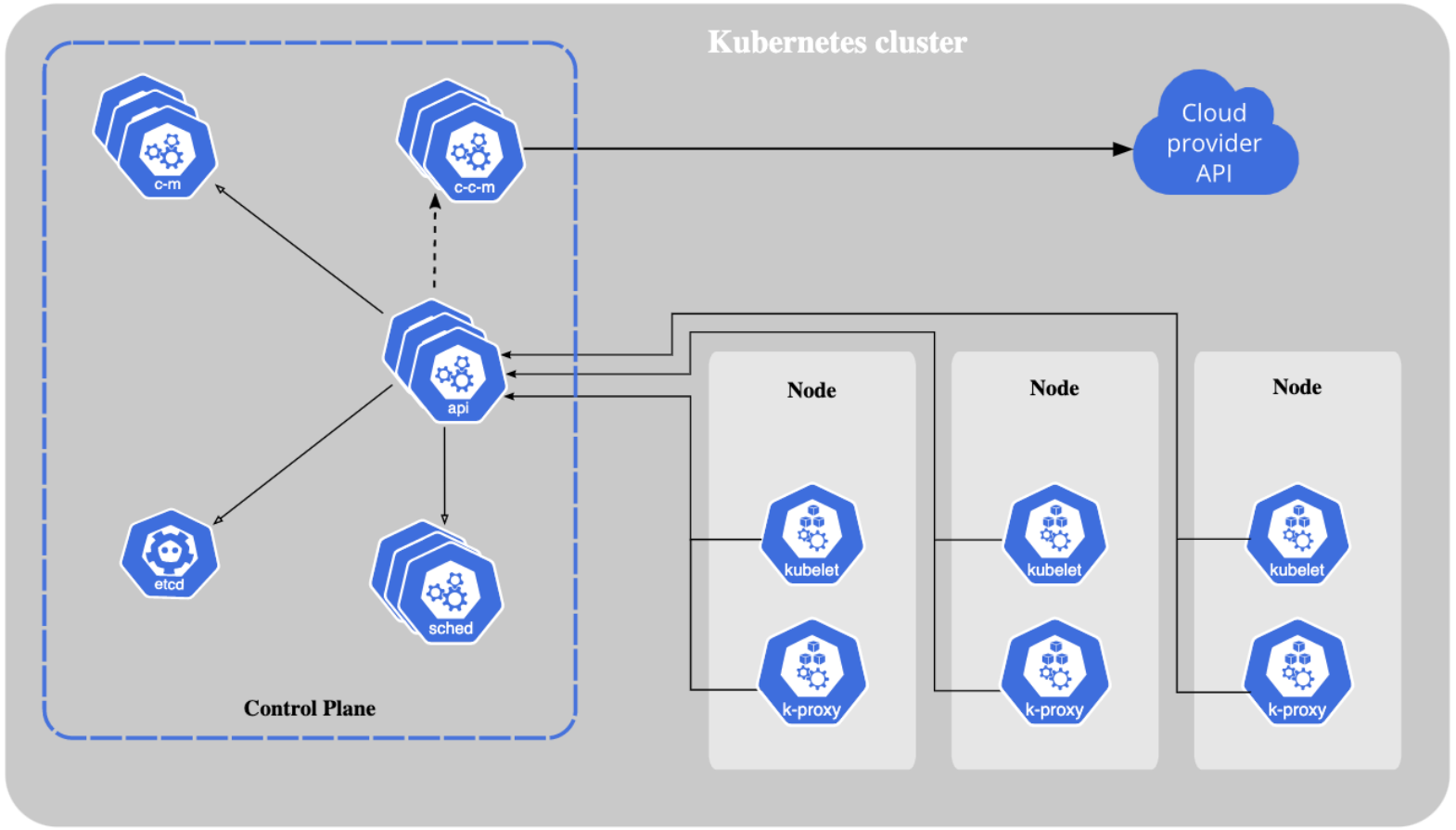
Concepts

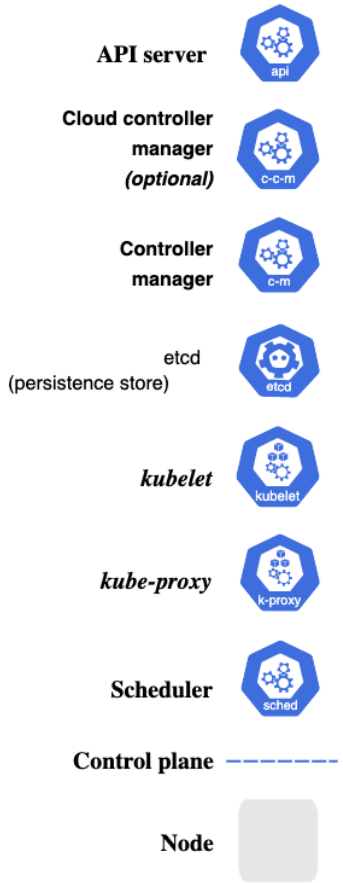
Implicit or Dynamic Grouping

- Explicit/Static Grouping
  - Group is defined by static list
  - Explicitly refers to object names
  - Not flexible and can't respond to dynamically changing world
- Dynamic/Implicit Grouping
  - Group doesn't explicitly refer to the members
  - Defines criteria for object selection
  - Achieve in K8S via labels and label queries/label selectors



Kubernetes Architecture





Kubernetes Architecture - Structure

- Unix philosophy of many components
  - K8S is a collection of different applications, every application ignorant of others which work together to implement the overall system
  - Even though there is a single binary (e.g. controller manager) implementing multiple functionalities
    - Every functionality is implemented independent of others
    - They are complied together to ease the deployment and management
    - Every components communicates with the API server rather than directly within running progress
  - Advantages of Modular Approach
    - Customization of Functionality
      - Large pieces of functionality can be ripped out and replaced without impacting rest of the system
  - Results into increased complexity

Kubernetes Architecture - Structure

- API-Driven Interactions
  - Interactions between components is driven through API server
  - No part of the system is more privileged or has more direct access to the internals (except of course the API server)
  - Allows customization of even core components without impacting other components e.g. scheduler

Kubernetes Architecture - Components

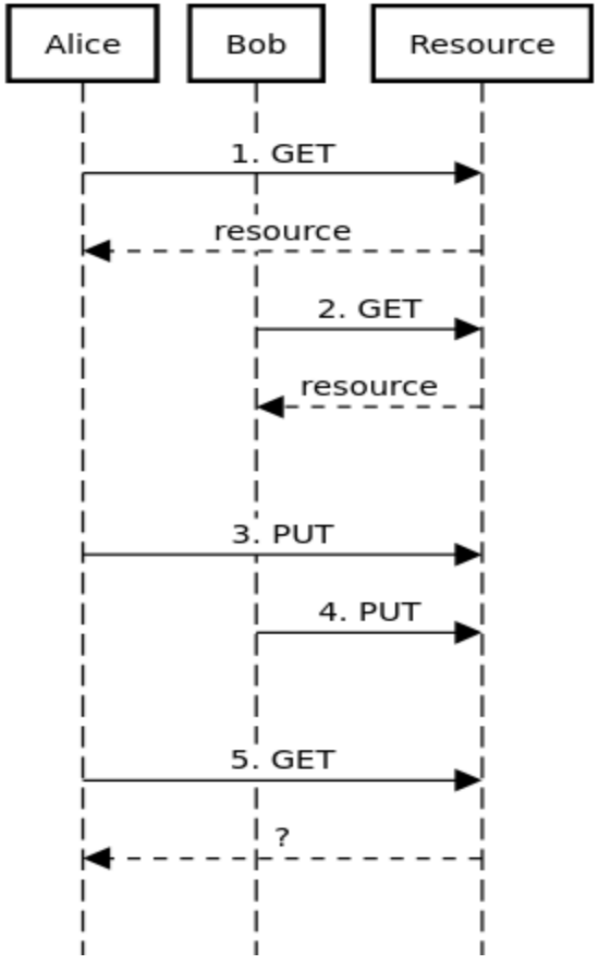
- Kubernetes is a system which groups a large fleet of machines into a single unit that can be consumed via an API
- Subdivides the machines in two groups
  - Worker Nodes
    - Host the pods serving application workload
    - Run selection of Kubernetes components
  - Master or Control Plane Nodes
    - Hosts control plane components
    - Limited number of such nodes in a cluster, generally one, three or five (to keep quorum in shared state using Raft/Paxos algorithm)

Kubernetes Architecture – Master Node Components

- Etcd
  - Implements key-value store to persist Kubernetes cluster objects
  - Change Notification
    - Implements “a watch” protocol, enabling the client to efficiently watch for changes in the key-value stores for an entire directory of values
    - Eliminates the need for continuous polling

Optimistic Concurrency

- Every value stored in *etcd* has resource version
- When a key-value pair is written, it can be conditionalized to specific resource version
- Allows to implement “compare and swap”, allowing concurrency
- Enables the system to have multiple threads manipulating data in etcd without pessimistic locks which reduces the throughput to a great extent



VERITAS

Kubernetes Architecture – Master Node Components

- API Server
  - Mediates all the interaction between clients and API objects stored in the etcd
- Scheduler
  - Scans the API server for unscheduled objects
  - Determines the best nodes to run the objects
- Controller Manager
  - Consists of reconciliation control loops for various functions
  - Ensure that desired declarative state is maintained

API Server

- Gateway to the Kubernetes Cluster
- Accessed by all users, components within the cluster
- Implements a RESTful API over HTTP
- Responsible for storing API objects into a persistent storage backend
- Kubernetes API Server involves three basic functions
  - API Management
  - Request Processing
  - Internal control loops

API Server

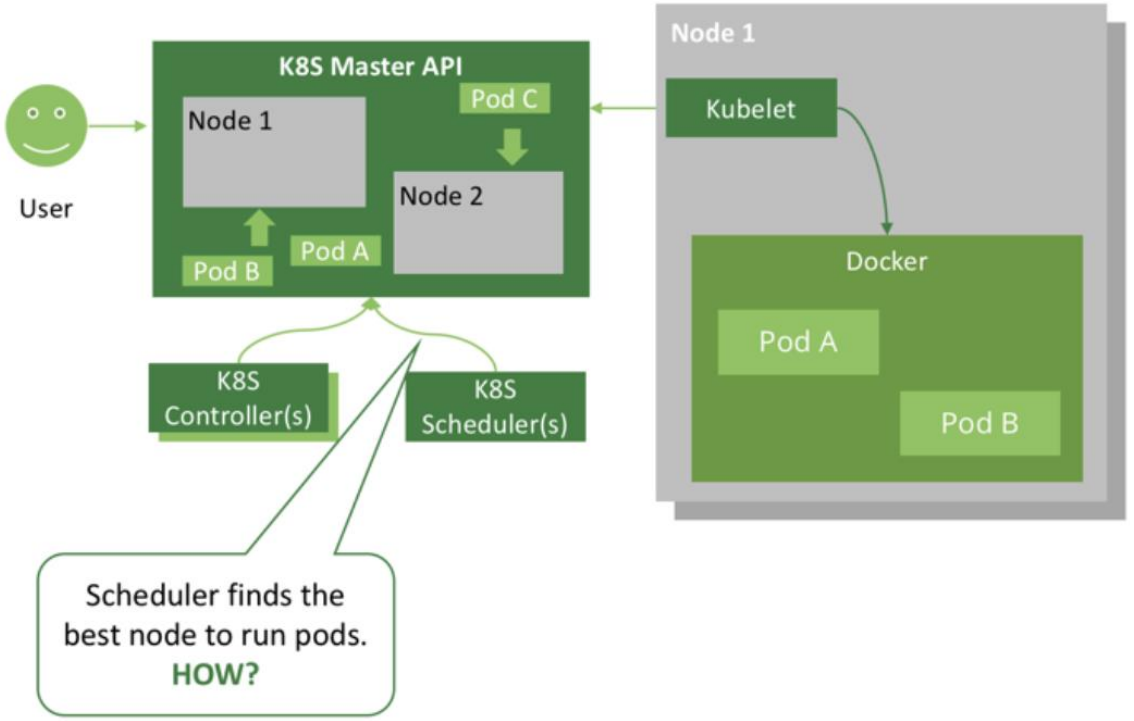
- API Management
  - API Paths
    - For every request, API server follows RESTful API pattern
    - All K8s requests begin with path /api/ (core APIs e.g. pod, service) and /apis/ (batch)
    - Component of Paths for *namespaced* resource
      - /api/v1/namespaces/<namespace-name>/<resource-type-name>/<resource-name>
      - /apis/<api-group>/<api-version>/namespaces/<namespace-name>/<resource-type-name>/<resource-name>
  - API Discovery
  - API Translation
    - API version lifecycle
      - v1alpha1
      - v1beta1
      - v1

API Server (Request Management)

- Request Types
  - GET, LIST, POST, DELETE
- Life of a Request
  - Authentication
  - RBAC/Authorization
  - Admission Control
    - Determines whether the request is well formed
    - Admission controllers are called serially (Mutating, first. Validating, second)
    - Pluggable mechanism
    - Examples
      - Add default values to objects
      - Enforce policy
      - Inject additional container into every pod

Scheduler

- Responsibilities
  - Watches for newly created Pods without any nodes assigned
  - Identifies the “best” node for scheduling the pod
- Scheduling Process
  - Filtering
    - Shortlists the node(s) based on the criteria
  - Scoring
    - Ranks the feasible nodes



Scheduler

- Predicates (for filtering nodes)
  - Indicates whether a pod fits onto a particular node
  - Hard constraints which if violated, lead to Pod not operating correctly
    - Pod Memory requirements
    - Node selector label query
- Priorities (for scoring nodes)
  - Generic interface used by scheduler to determine preference of one node over another
  - Priority Functions
    - Assumes that pod can be scheduled onto the node
    - Score relative value of scheduling a pod onto a particular node
      - Weigh nodes where image has already been pulled
      - Spreading Function
        - Prioritizes nodes where pods that are members of same Kubernetes service are not present
    - Predicate values are mixed together to achieve final priority score for a node
  - Node with best score is selected for pod scheduling

Kubernetes Architecture – Components on all nodes

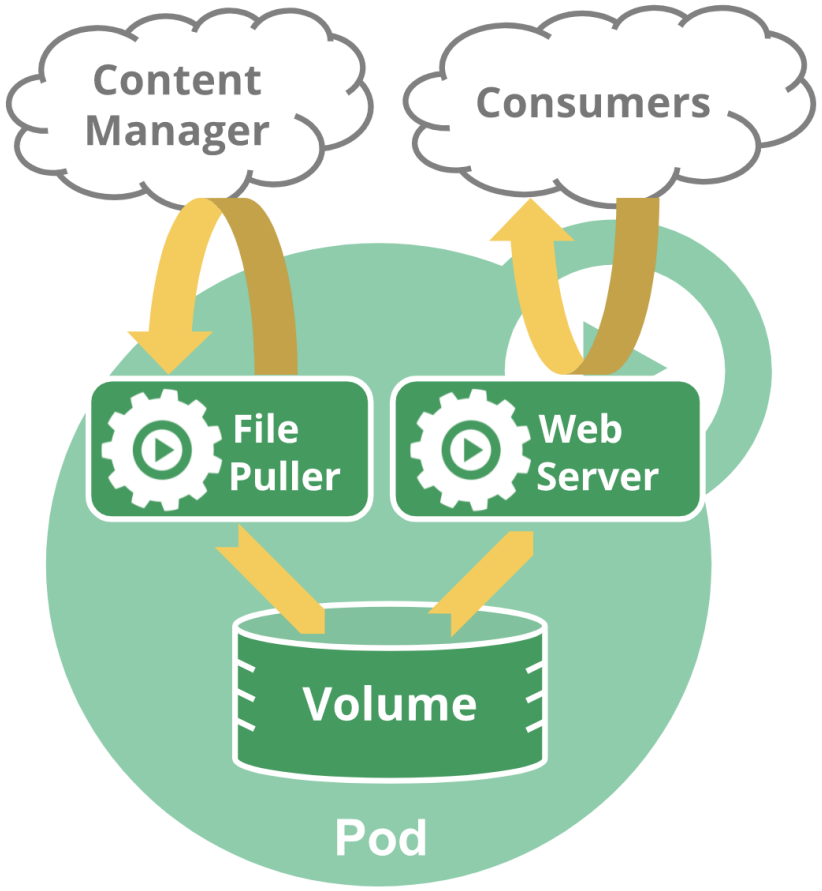
- Kubelet
  - Node daemon for all machines that are part of K8S cluster
  - A bridge that joins available CPU, disk and memory for a node into large K8S cluster
  - Scheduling and Reporting
    - From API Server, retrieves the information about pods schedule to run on specific node
    - Communicates the state of pods running on the node to the API server
      - Enables the other reconciliation loops to observe the current state of pods
  - Health Check and Healing
    - Performs health checks and restarts the containers running on the node
    - Instead of pushing the health-state information to API server and let reconciliation loops take action to fix the health, the ”local” healing is more efficient

Kubernetes Architecture – Components on all nodes

- Kube-proxy
  - Every service in Kubernetes gets a virtual IP address
  - Kube-proxy is responsible for implementing the Kubernetes service load-balancer networking model
  - Watches the endpoint objects for all services in the K8S cluster
  - Programs the network on its node so that all network requests to the virtual IP address of a service are routed to the endpoints that implement the service

Kubernetes Architecture: Pods

- Collection of one or more containers
- Atomic unit of scheduling in Kubernetes cluster
  - All containers in the pod are guaranteed to run on the same node
- Pods share resource between containers
  - Pods share process and interprocess communication namespaces
  - Pods share same network namespace
- In a typical deployment, pod consists of single container
- Multiple containers example – sidecar container
- Health Checks and Self Healing



VERITAS

Kubernetes Architecture: Services

- Every service gets three things
  - It's own IP address (virtual)
  - DNS entry in Kubernetes cluster DNS
  - Load-balancing rules that proxy traffic to the Pods that implement the Service
- Load Balancing
  - Service load balancing is programmed into network fabric of Kubernetes cluster's DNS server
  - Any container that tries to talk to the Service IP address is correctly balanced to corresponding pods
  - Dynamic update of network fabric as pods are scaled out/in
  - Clients can rely on Service IP address to resolve to a Pod which implements the service

Kubernetes Architecture: ReplicaSets

- Horizontal Scaling
  - Spin up multiple instances of an application
  - Allows to grow application in response to the load
- *Replicaset* ensures that for a given pod definition, number of replicas exist within the system
- Kubernetes controller manager runs reconciliation loop to handle replication

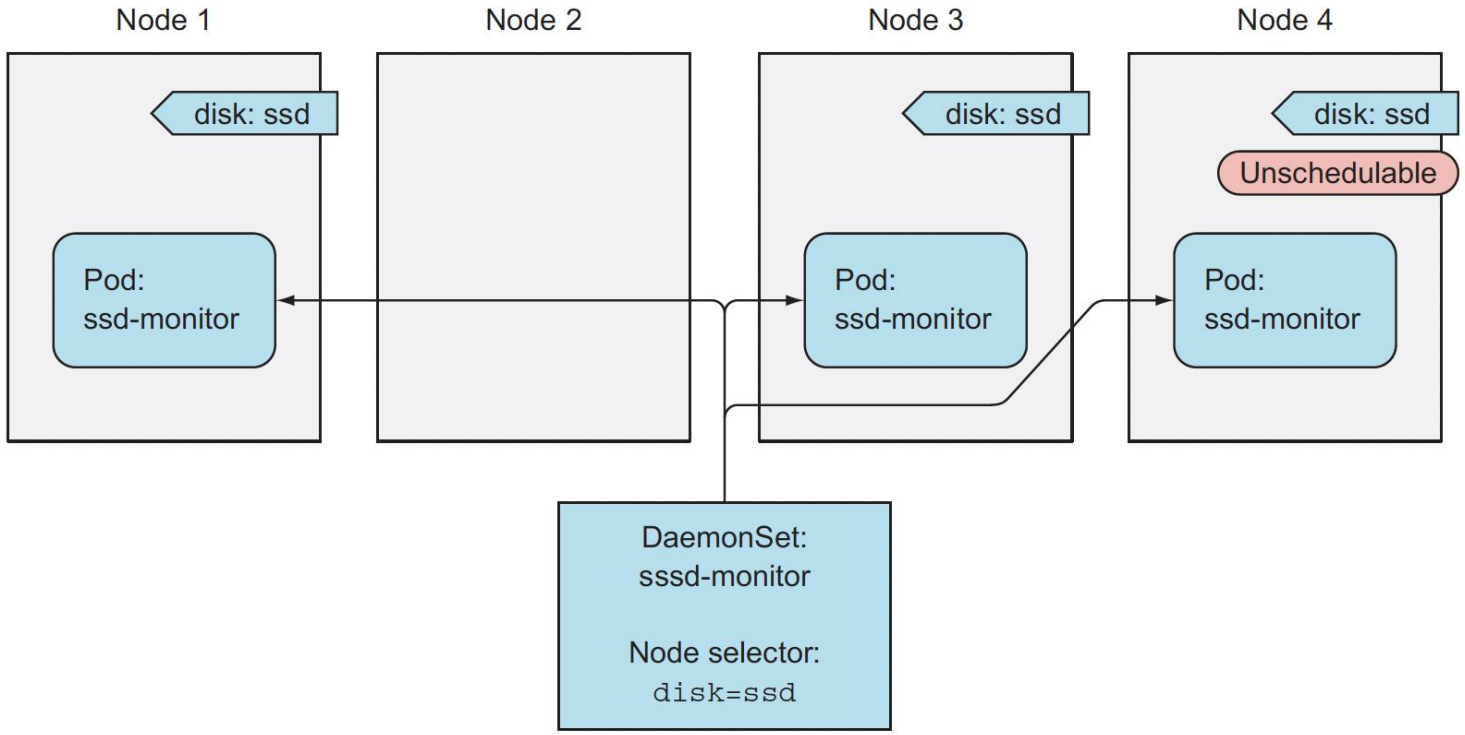
ReplicaSet Spec

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
```

DaemonSets

- Why?
  - To run exactly one pod per node. Ex: system daemons.
  - To bypass the scheduler. Ex: system daemons.

DaemonSets



Let's imagine having a daemon called ssd-monitor that needs to run on all nodes that contain a solid-state drive (SSD). You'll create a DaemonSet that runs this daemon on all nodes that are marked as having an SSD. The cluster administrators have added the disk=ssd label to all such nodes, so you'll create the DaemonSet with a node selector that only selects nodes with that label, as shown in the figure to the left.

DaemonSets

A YAML for a DaemonSet: ssd-monitor-daemonset.yaml

```
apiVersion: apps/v1beta2
kind: DaemonSet
metadata:
  name: ssd-monitor
spec:
  selector:
    matchLabels:
      app: ssd-monitor
  template:
    metadata:
      labels:
        app: ssd-monitor
    spec:
      nodeSelector:
        disk: ssd
      containers:
      - name: main
        image: luksa/ssd-monitor
```

DaemonSets are in the apps API group, version v1beta2.

The pod template includes a node selector, which selects nodes with the disk=ssd label.

VERITAS

Batch Jobs

- Why?
  - To run ad-hoc tasks immediately, rather than implement continuously running processes.
    - Tasks end, Processes aren't supposed to.
- The Job resource.
  - After pod reaches completed state, its not restarted unlike regular pods.
  - In event of node failure, the Job will get rescheduled to another node like ReplicaSets.
  - In event of process failure, the Job can be configured to either restart the container or not.

Batch Jobs

A YAML definition of a Job: exporter.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: batch-job
spec:
  template:
    metadata:
      labels:
        app: batch-job
    spec:
      restartPolicy: OnFailure
      containers:
      - name: main
        image: luksa/batch-job
```

Jobs are in the batch API group, version v1.

You're not specifying a pod selector (it will be created based on the labels in the pod template).

Jobs can't use the default restart policy, which is Always.

VERITAS

CronJobs

- Why?
  - To run ad-hoc tasks at scheduled times, rather than implement continuously running processes or to run them immediately like batch jobs.
- The CronJob resource.
  - A CronJob is a Job.
  - The schedule for running the Job is specified in the well-known cron format.
  - After pod reaches completed state, its not restarted unlike regular pods.
  - In event of node failure, the CronJob will get rescheduled to another node like ReplicaSets.
  - In event of process failure, the CronJob can be configured to either restart the container or not.

CronJobs

Listing 4.14 YAML for a CronJob resource: cronjob.yaml

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: batch-job-every-fifteen-minutes
spec:
  schedule: "0,15,30,45 * * * *"
  jobTemplate:
    spec:
      template:
        metadata:
          labels:
            app: periodic-batch-job
        spec:
          restartPolicy: OnFailure
          containers:
          - name: main
            image: luksa/batch-job
```

API group is batch, version is v1beta1

This job should run at the 0, 15, 30 and 45 minutes of every hour, every day.

The template for the Job resources that will be created by this CronJob

Scheduled Jobs run at *approximately* the times specified in the schedule. If we want the Jobs to have a maximum amount of time tolerance for our Job runs, we ought to specify a “startingDeadlineSeconds” value in the Job spec.

Listing 4.15 Specifying a `startingDeadlineSeconds` for a CronJob

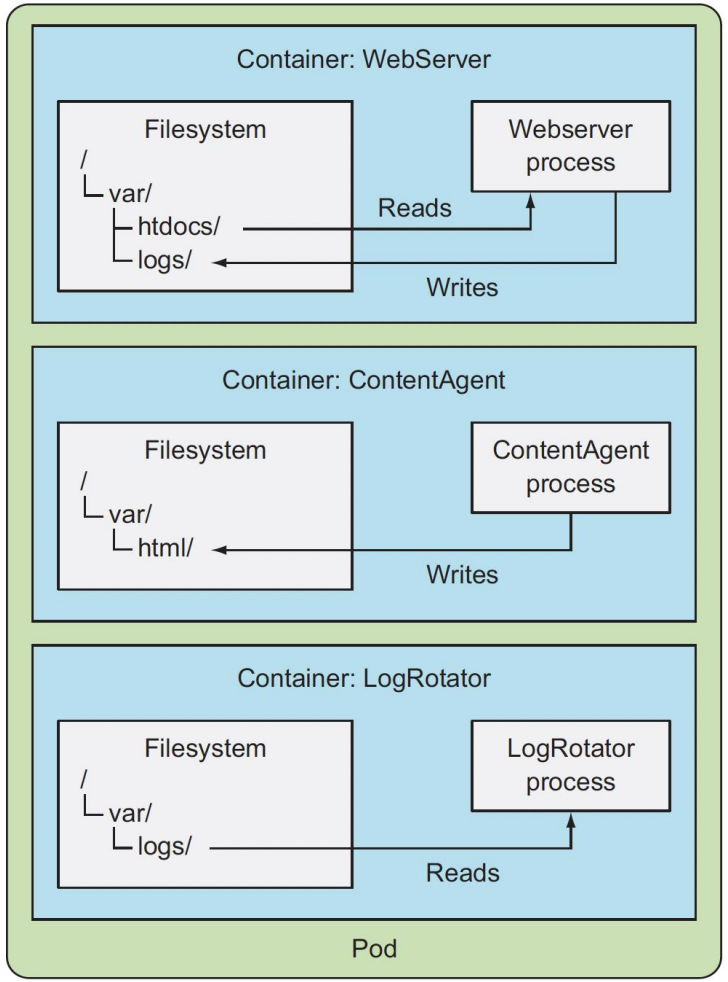
```
apiVersion: batch/v1beta1
kind: CronJob
spec:
  schedule: "0,15,30,45 * * * *"
  startingDeadlineSeconds: 15
  ...
```

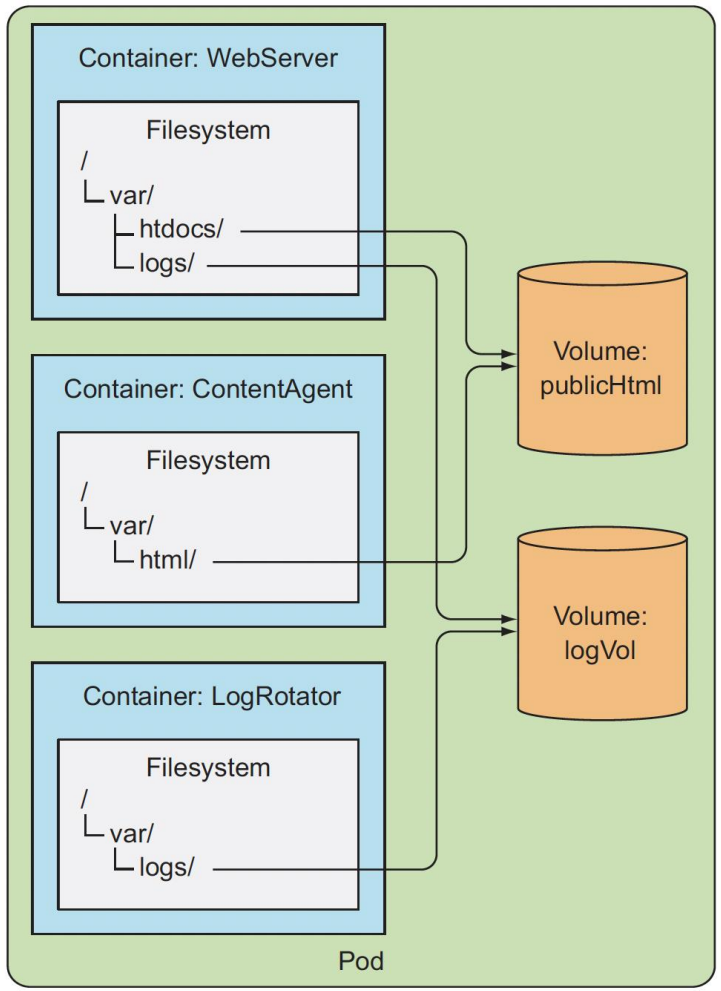
At the latest, the pod must start running at 15 seconds past the scheduled time.

VERITAS

Kubernetes Storage Concepts

Volumes





- Why?
  - To make files persist across pod lifecycles.

Types of Volumes (not an exhaustive list)

- emptyDir
- gitRepo
- hostPath
- Nfs
- gcePersistentDisk (GCP)
- awsElasticBlockStore (AWS EBS)
- azureDisk (Azure disk volume)
- cinder
- cephfs
- iscsi
- glusterfs
- vsphere-Volume
- configMap
- secret
- persistentVolumeClaim

emptyDir Volume

A pod with two containers sharing the same volume: fortune-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: fortune
spec:
  containers:
  - image: luksa/fortune
    name: html-generator
    volumeMounts:
    - name: html
      mountPath: /var/htdocs
  - image: nginx:alpine
    name: web-server
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
      readOnly: true
    ports:
    - containerPort: 80
      protocol: TCP
  volumes:
  - name: html
    emptyDir: {}
```

The first container is called html-generator and runs the luksa/fortune image.

The volume called html is mounted at /var/htdocs in the container.

The second container is called web-server and runs the nginx:alpine image.

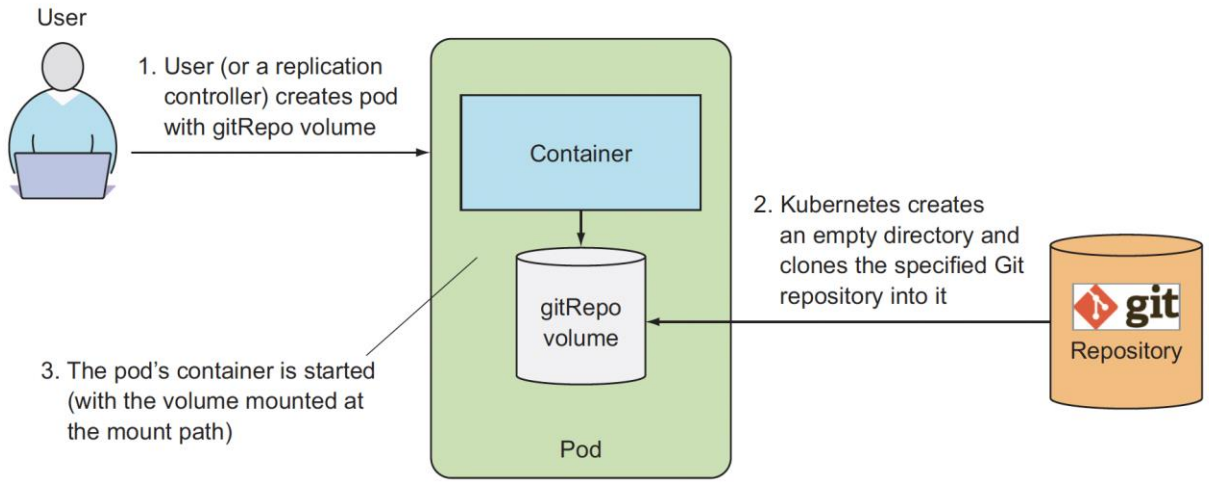
The same volume as above is mounted at /usr/share/nginx/html as read-only.

A single emptyDir volume called html that's mounted in the two containers above

The volume starts out as an empty directory. The app running inside the pod writes files to it. The files are written to disk on the worker node. When the pod is deleted, the volume is deleted too. The contents of the volume persist across pod restarts, though. Especially useful for sharing files between containers within the same pod.

VERITAS

gitRepo Volume



A `gitRepo` volume is an `emptyDir` volume initially populated with the contents of a Git repository.

A gitRepo volume is basically an emptyDir volume that gets populated by cloning a Git repository and checking out a specific revision when the pod is starting up (but before the containers are created).

gitRepo Volume

A pod using a `gitRepo` volume: gitrepo-volume-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: gitrepo-volume-pod
spec:
  containers:
  - image: nginx:alpine
    name: web-server
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
      readOnly: true
    ports:
    - containerPort: 80
      protocol: TCP
  volumes:
  - name: html
    gitRepo:
      repository: https://github.com/luksa/kubia-website-example.git
      revision: master
      directory: .
```

You're creating a gitRepo volume.

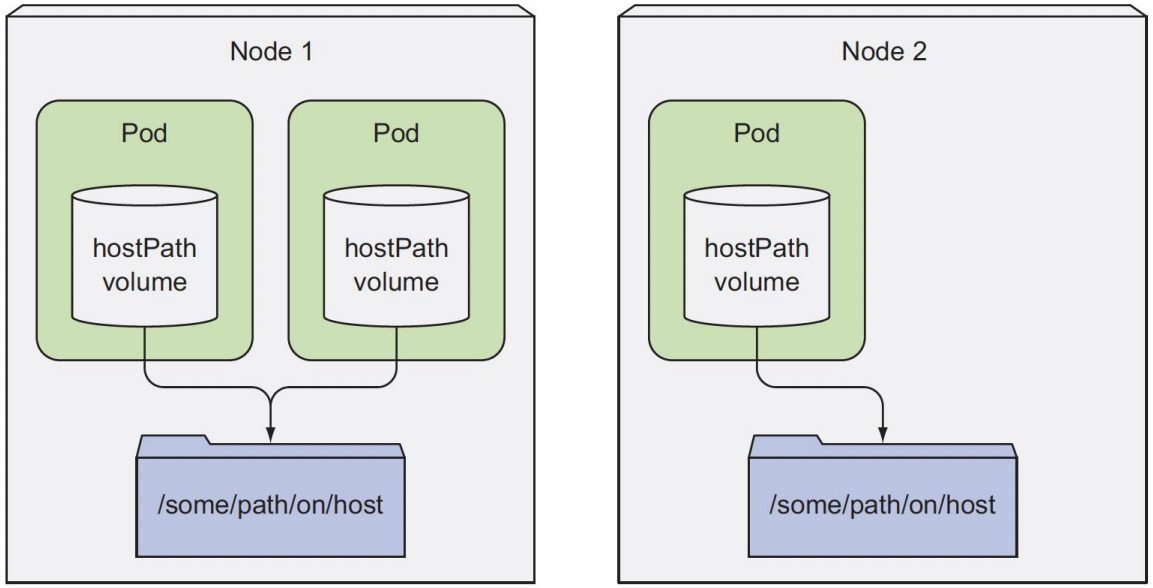
The volume will clone this Git repository.

The master branch will be checked out.

You want the repo to be cloned into the root dir of the volume.

VERITAS

hostPath Volume



A `hostPath` volume mounts a file or directory on the worker node into the container's filesystem.

A hostPath volume points to a specific file or directory on the node's filesystem. Pods running on the same node and using the path in their hostPath volume see the same files. Unlike the previous types, hostPath volumes are truly persistent and outlast the pod lifecycles.

```
$ kubectl describe po fluentd-kubia-4ebc2f1e-9a3e --namespace kube-system
Name:           fluentd-cloud-logging-gke-kubia-default-pool-4ebc2f1e-9a3e
Namespace:      kube-system
...
Volumes:
  varlog:
    Type:       HostPath (bare host directory volume)
    Path:       /var/log
  varlibdockercontainers:
    Type:       HostPath (bare host directory volume)
    Path:       /var/lib/docker/containers
```

Decoupling pods from storage technology

All the persistent volume types we have discussed so far have required the developer of the pod to have knowledge of the actual network storage infrastructure available.

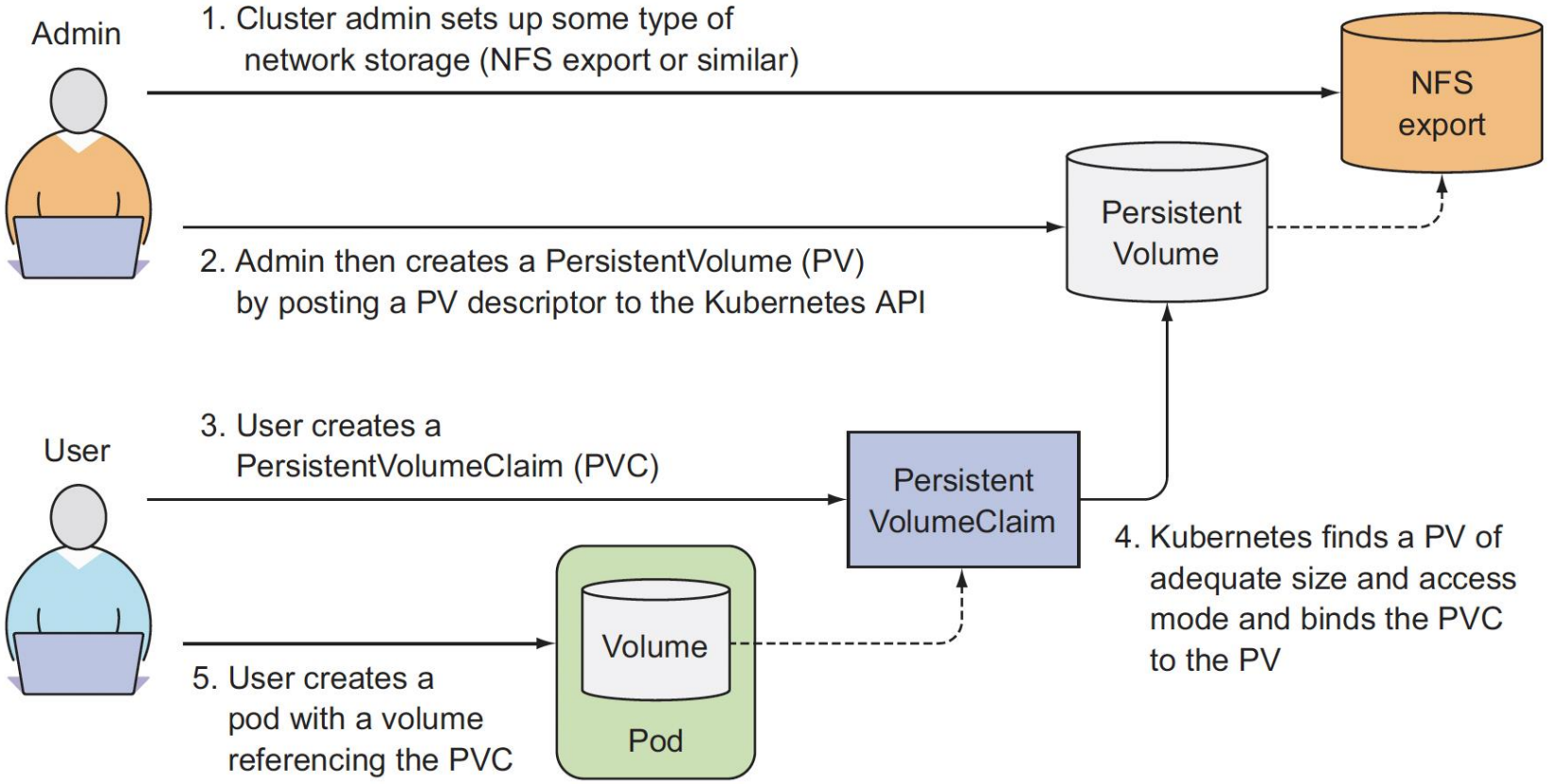
This goes against the basic idea of Kubernetes, which aims to hide the actual infrastructure from both the application and its developer.

Ideally, a developer deploying apps on Kubernetes should never have to know what kind of storage technology is used underneath. Ideally, storage administration should be a part of cluster administration, rather than development.

To enable apps to request storage in a Kubernetes cluster without having to deal with infrastructure specifics, two new resources are now going to be introduced:

- Persistent Volumes (PV)
- Persistent Volume Claims (PVC)

Persistent Volumes



PersistentVolumes are provisioned by cluster admins and consumed by pods through PersistentVolumeClaims.

Provisioning Persistent Volumes from GCP storage (example)

A `gcePersistentDisk` PersistentVolume: mongodb-pv-gcepd.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongodb-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
  - ReadWriteOnce
  - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  gcePersistentDisk:
    pdName: mongodb
    fsType: ext4
```

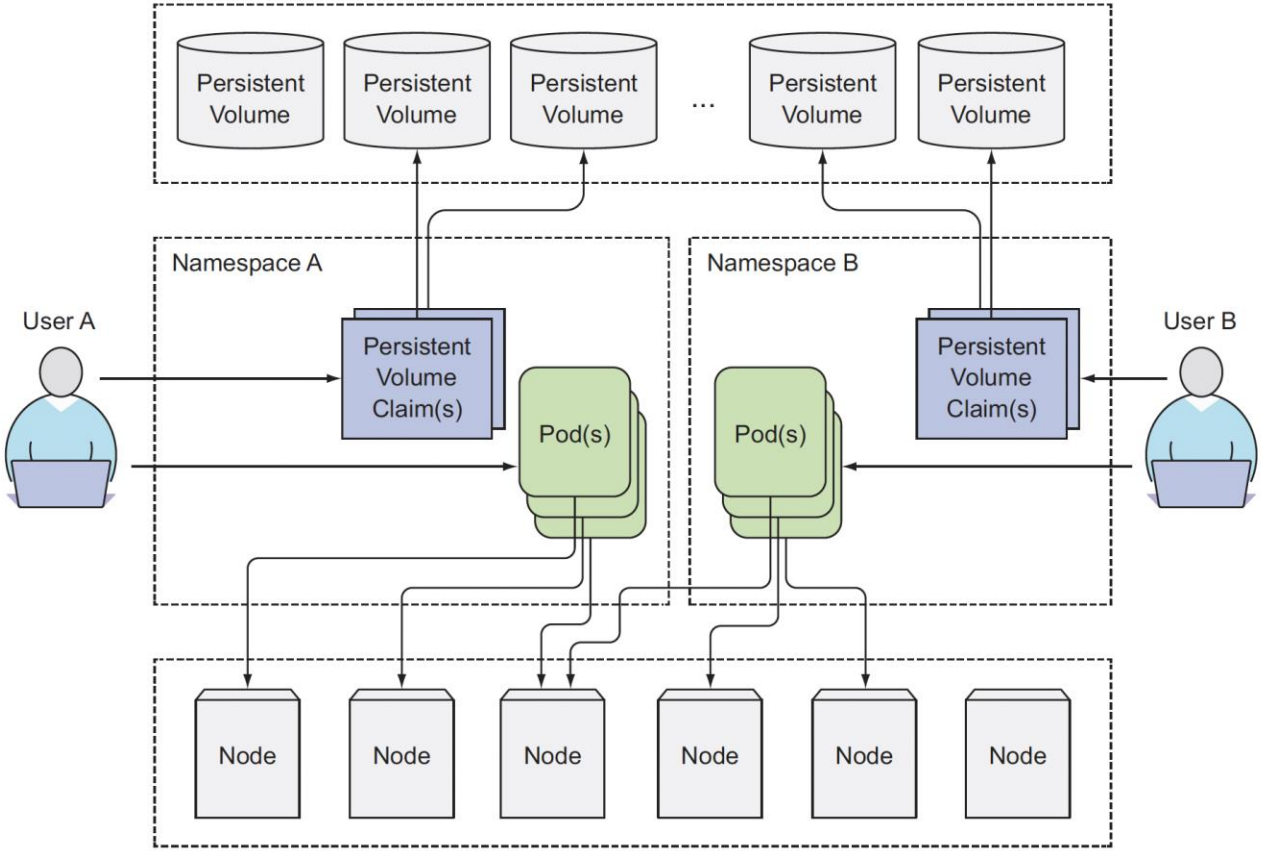
Defining the PersistentVolume's size

It can either be mounted by a single client for reading and writing or by multiple clients for reading only.

After the claim is released, the PersistentVolume should be retained (not erased or deleted).

The PersistentVolume is backed by the GCE Persistent Disk you created earlier.

PersistentVolumes (PV), a cluster-level resource



PersistentVolumes, like cluster Nodes, don't belong to any namespace, unlike pods and PersistentVolumeClaims.

PersistentVolumesClaims (PVC)

A `PersistentVolumeClaim`: mongodb-pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodb-pvc
spec:
  resources:
    requests:
      storage: 1Gi
  accessModes:
  - ReadWriteOnce
  storageClassName: ""
```

The name of your claim—you'll need this later when using the claim as the pod's volume.

Requesting 1 GiB of storage

You want the storage to support a single client (performing both reads and writes).

You'll learn about this in the section about dynamic provisioning.

A Persistent Volume Claim is completely different from a Persistent Volume. A Claim is supposed to stay available even if the pod is deleted or moved, so that it can be attached to a new pod taking the place of the old pod. As soon as a PVC is created, Kubernetes finds the appropriate PersistentVolume and binds it to the claim. The PersistentVolume's capacity must be large enough to accommodate the claim request. Additionally, the volume's access modes must include the access modes requested by the claim.

VERITAS

Using a claimed PersistentVolume inside a pod

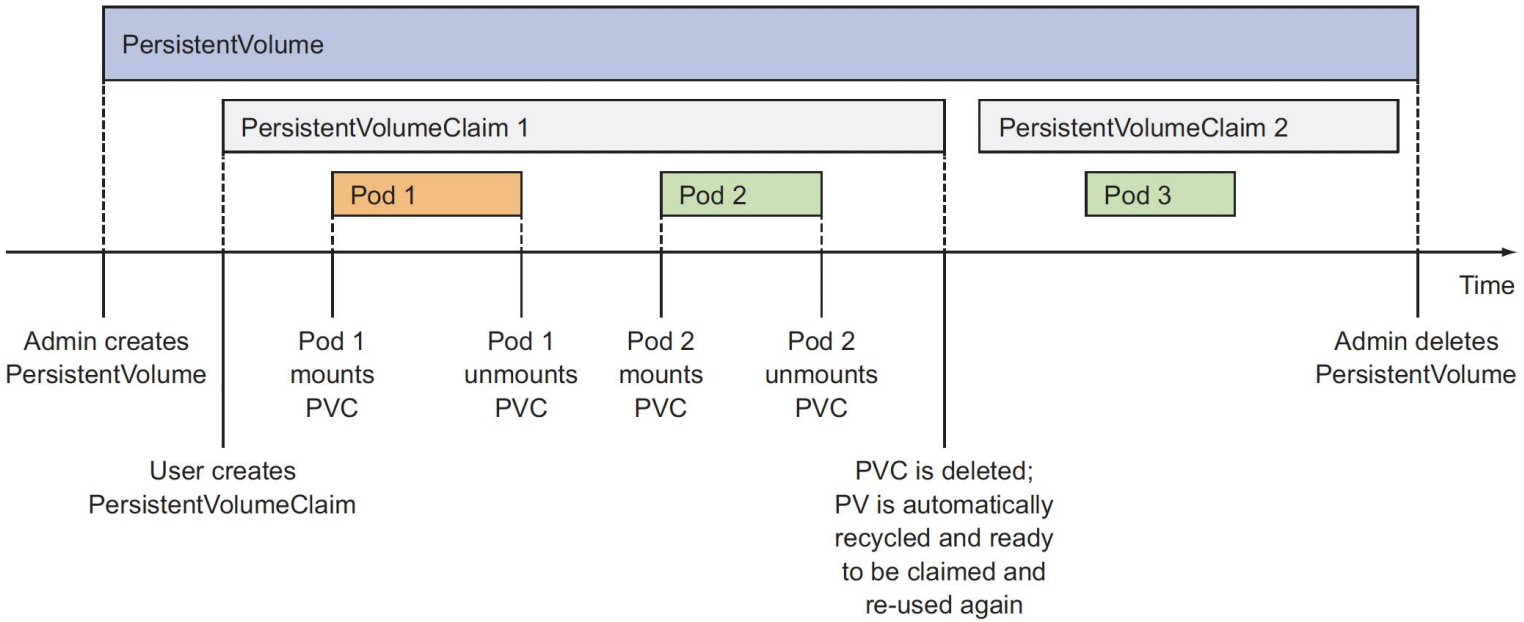
A pod using a PersistentVolumeClaim volume: mongodb-pod-pvc.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: mongodb
spec:
  containers:
  - image: mongo
    name: mongodb
    volumeMounts:
    - name: mongodb-data
      mountPath: /data/db
    ports:
    - containerPort: 27017
      protocol: TCP
  volumes:
  - name: mongodb-data
    persistentVolumeClaim:
      claimName: mongodb-pvc
```

Referencing the PersistentVolumeClaim by name in the pod volume

To use the PersistentVolume bound to your PersistentVolumeClaim within your pod, you need to reference the PersistentVolumeClaim by name inside the pod's volume listings as shown.

PVC reclaim policies



The lifespan of a PersistentVolume, PersistentVolumeClaims, and pods using them

In our example, we set persistentVolumeReclaimPolicy to Retain when creating the PV. What this means is that once the PVC that claimed a PV is deleted, its contents will be preserved – surviving across different PVCs. Two other possible reclaim policies exist: Recycle and Delete. The former deletes the contents of the volume and makes an empty volume available for the next PVC, whereas the latter deletes the underlying storage.

Dynamic Provisioning

In the examples so far, we saw how the storage administrator can provision storage for pods, and the developers can use this provisioned storage in their pods. But in today's world of DevOps, often the developer would perform the duties of a storage admin. In this case, it would be better to have Kubernetes provision storage for the pods as well, to ease the burden of manual work on the Developer.

This can be done by Kubernetes through dynamic provisioning of PersistentVolumes:

- The Developer/Storage admin can deploy a PersistentVolume provisioner.
- Then, define one or more StorageClass objects to let users choose what type of PersistentVolume they want.
- The users can refer to the StorageClass in their PersistentVolumeClaims and the provisioner will take that into account when provisioning the persistent storage.

Kubernetes includes provisioners for the most popular cloud providers, but if Kubernetes is deployed on-premises, a custom provisioner needs to be deployed.

Defining a StorageClass object

A StorageClass definition: storageclass-fast-gcepd.yaml

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
  zone: europe-west1-b
```

The volume plugin to use for provisioning the PersistentVolume

The parameters passed to the provisioner

The StorageClass resource specifies which provisioner should be used for provisioning the PersistentVolume when a PersistentVolumeClaim requests this StorageClass. The parameters defined in the StorageClass definition are passed to the provisioner and are specific to each provisioner plugin.

VERITAS

Requesting a storage class in a PersistentVolumeClaim

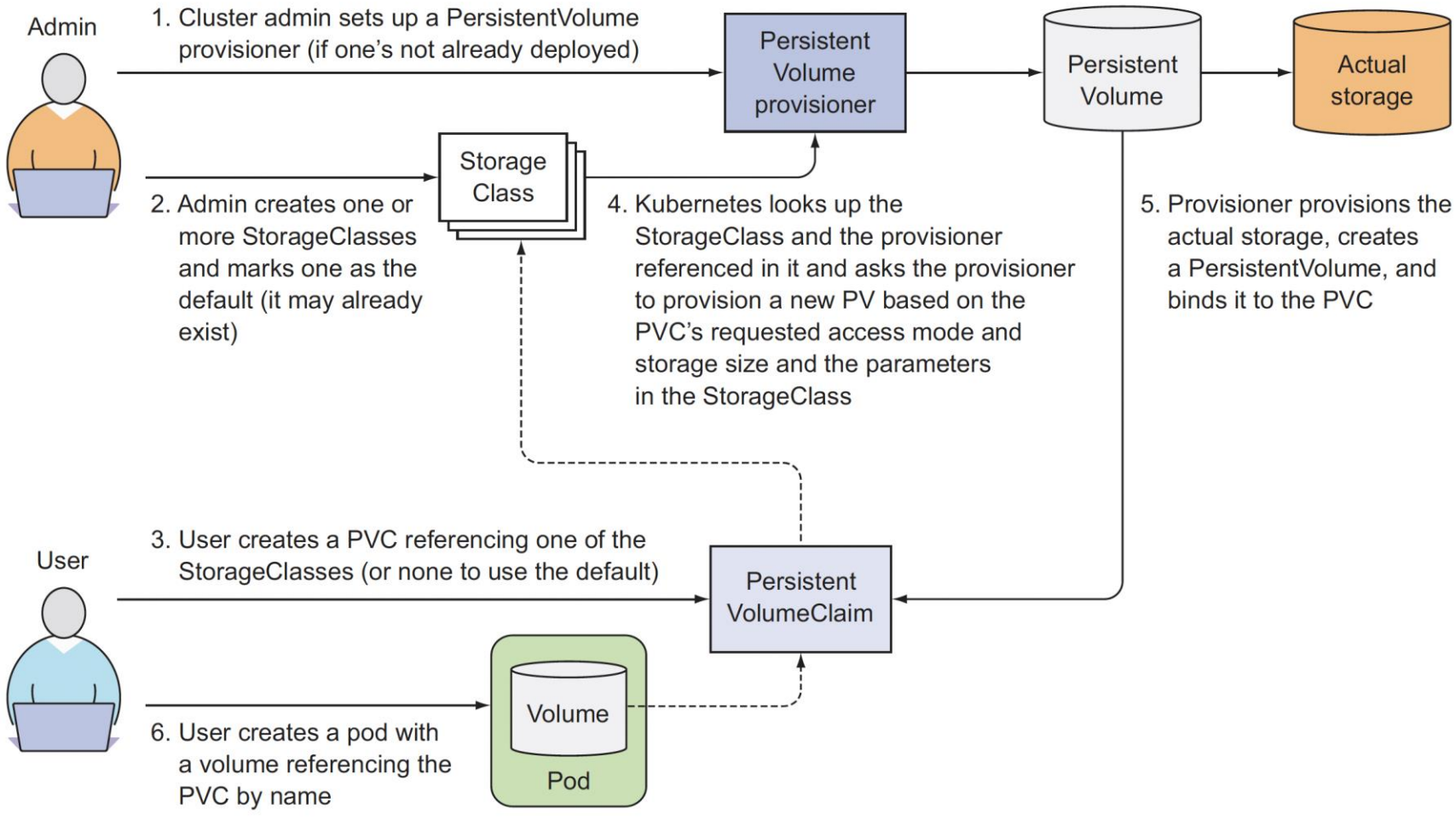
A PVC with dynamic provisioning: mongodb-pvc-dp.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodb-pvc
spec:
  storageClassName: fast
  resources:
    requests:
      storage: 100Mi
  accessModes:
    - ReadWriteOnce
```

This PVC requests the custom storage class.

The PersistentVolumeClaim now specifies the name of the StorageClass you want to use. When the PVC is created, the PV is created by the provisioner referenced in the StorageClass resource. The provisioner is used even if an existing manually provisioned PV matches the PVC. Typically, you would have a "default" StorageClass defined for Dynamic provisioning on your Kubernetes cluster. If no StorageClass is referenced in a PVC, the PV is created by the provisioner referenced in the default StorageClass for that cluster.

Complete picture of dynamic provisioning of PersistentVolumes (illustration)



ConfigMaps

YAML definition of a config map created from a file

```
$ kubectl get configmap fortune-config -o yaml
apiVersion: v1
data:
  my-nginx-config.conf: |
    server {
      listen              80;
      server_name         www.kubia-example.com;

      gzip on;
      gzip_types text/plain application/xml;

      location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
      }
    }
  sleep-interval: |
    25
kind: ConfigMap
...
```

The entry holding the Nginx config file's contents

The sleep-interval entry

NOTE The pipeline character after the colon in the first line of both entries signals that a literal multi-line value follows.

A ConfigMap is a Kubernetes object which contains key/value pairs that are used to specify the environment variables/config values for processes running in containers within pods.

What are the different ways of specifying environment variables for containers in pods apart from using ConfigMaps? – Homework Question.

VERITAS

How to create ConfigMaps?

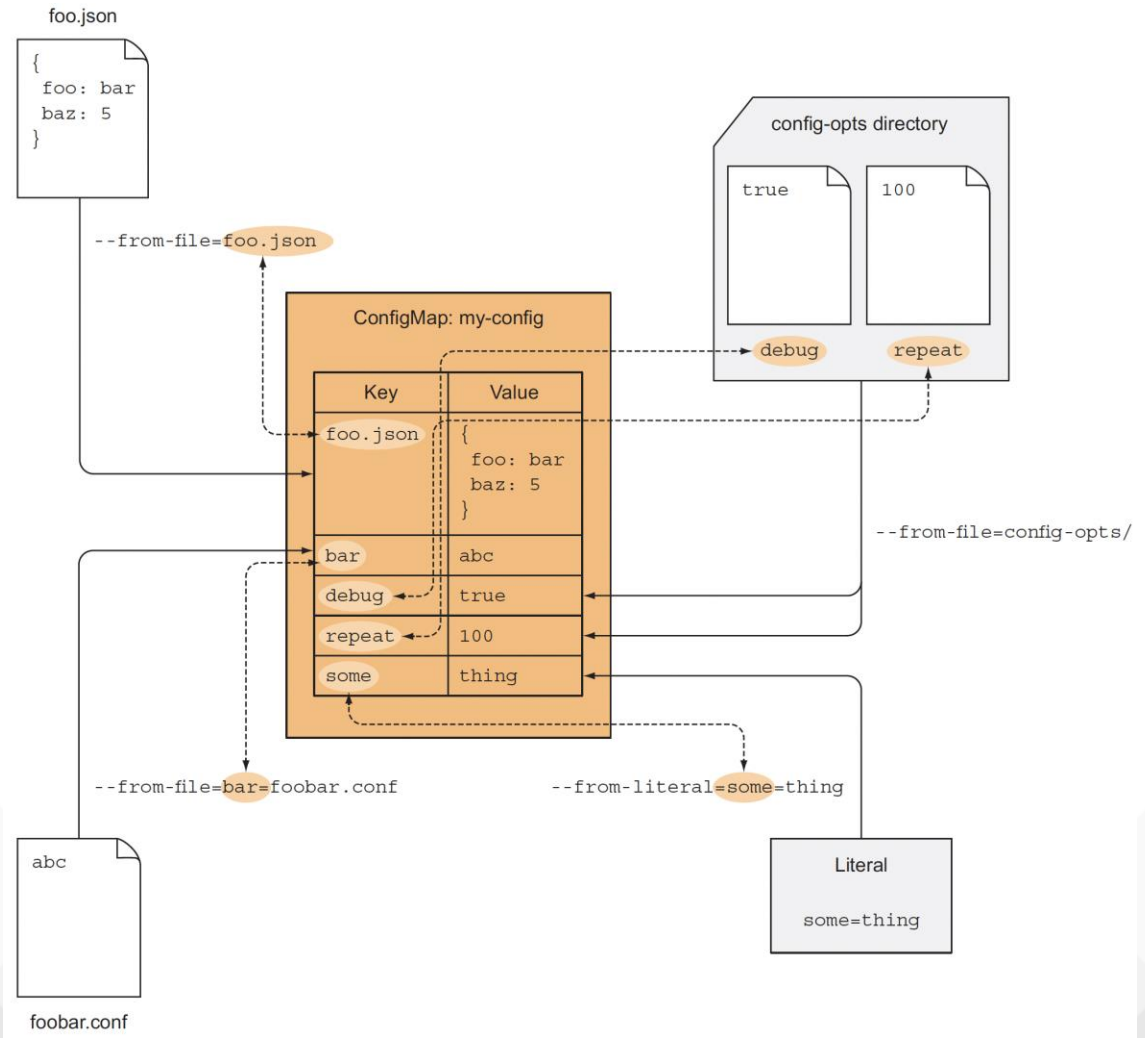
```
$ kubectl create configmap my-config
    --from-file=foo.json
    --from-file=bar=foobar.conf
    --from-file=config-opts/
    --from-literal=some=thing
```

A single file

A file stored under a custom key

A whole directory

A literal value



VERITAS

How to use ConfigMaps with pods?

A pod with ConfigMap entries mounted as files: fortune-pod-configmap-volume.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: fortune-configmap-volume
spec:
  containers:
  - image: nginx:alpine
    name: web-server
    volumeMounts:
    ...
    - name: config
      mountPath: /etc/nginx/conf.d
      readOnly: true
    ...
  volumes:
  ...
  - name: config
    configMap:
      name: fortune-config
  ...
```

You're mounting the configMap volume at this location.

The volume refers to your fortune-config ConfigMap.

This pod definition includes a volume, which references our ConfigMap. This volume is mounted into the /etc/nginx/conf.d directory inside the container to make nginx use it. Note that we are not creating the volume separately, either manually or using dynamic provisioning. The volume is a special volume that is populated with our config parameters in the same form which we provided earlier to create the config-map.

VERITAS

Secrets

A Secret's YAML definition

```
$ kubectl get secret fortune-https -o yaml
apiVersion: v1
data:
  foo: YmFyCg==
  https.cert: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURCekNDQ...
  https.key: LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSUlFcE...
kind: Secret
...
```

A ConfigMap's YAML definition

```
$ kubectl get configmap fortune-config -o yaml
apiVersion: v1
data:
  my-nginx-config.conf: |
    server {
      ...
    }
  sleep-interval: |
    25
kind: ConfigMap
...
```

Secrets are a lot like ConfigMaps. They are also Kubernetes objects which contains key/value pairs. The difference is that they key/value pairs are usually sensitive information, credentials, private encryption keys, etc – things which need to be kept secure.

Another difference between ConfigMaps and Secrets is that data values specified in Secrets are usually encoded. In the picture to the left, the Secret's entries are encoded in Base64-encoded strings. Kubernetes supports encoding/decoding of strings for Secrets, so that Secrets can be used to pass binary data encoded in plain-text format.

VERITAS

How to use Secrets with pods?

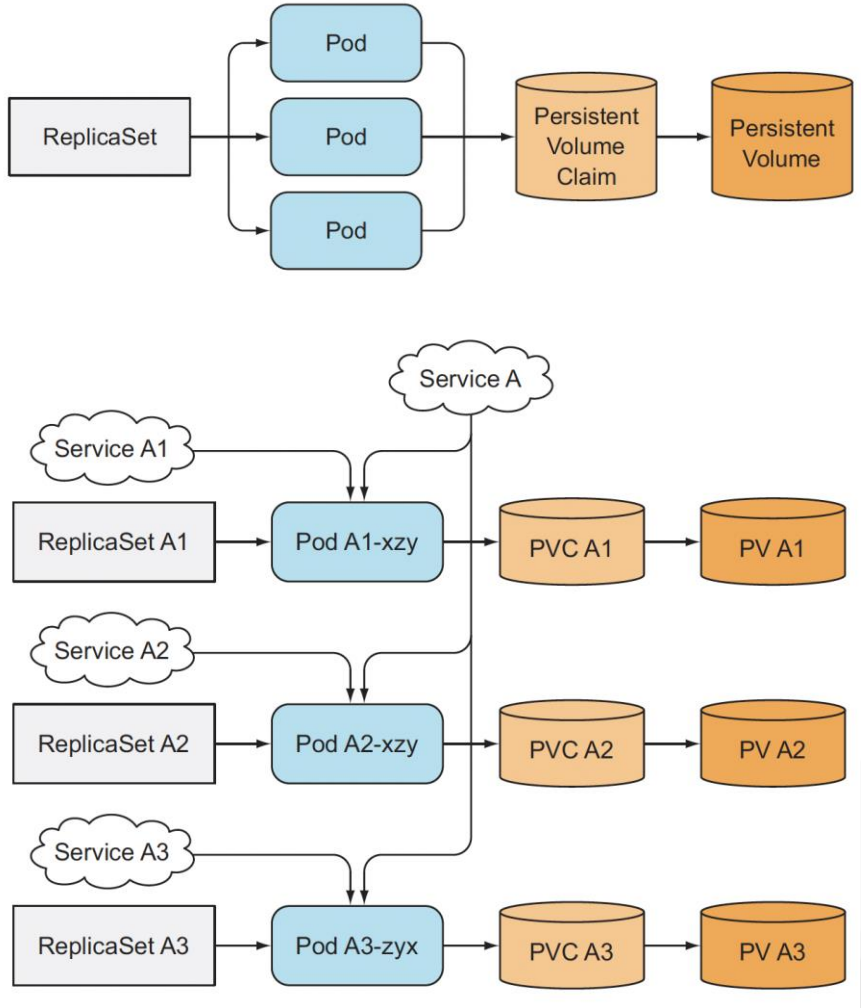
YAML definition of the `fortune-https` pod: fortune-pod-https.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: fortune-https
spec:
  containers:
  - image: luksa/fortune:env
    name: html-generator
    env:
    - name: INTERVAL
      valueFrom:
        configMapKeyRef:
          name: fortune-config
          key: sleep-interval
    volumeMounts:
    - name: html
      mountPath: /var/htdocs
  - image: nginx:alpine
    name: web-server
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
      readOnly: true
    - name: config
      mountPath: /etc/nginx/conf.d
      readOnly: true
    - name: certs
      mountPath: /etc/nginx/certs/
      readOnly: true
    ports:
    - containerPort: 80
    - containerPort: 443
  volumes:
  - name: html
    emptyDir: {}
  - name: config
    configMap:
      name: fortune-config
      items:
      - key: my-nginx-config.conf
        path: https.conf
  - name: certs
    secret:
      secretName: fortune-https
```

You configured Nginx to read the cert and key file from /etc/nginx/certs, so you need to mount the Secret volume there.

You define the secret volume here, referring to the fortune-https Secret.

The problem of diversity in unity. A/P deployments.

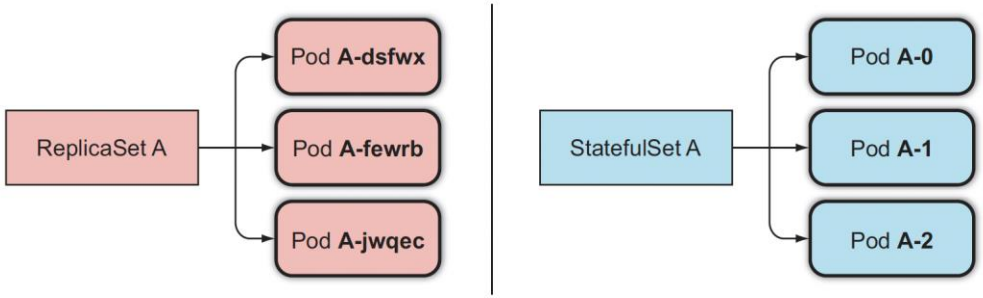


ReplicaSets create multiple pod replicas from a single pod template. These replicas don't differ from each other, apart from their name and IP address. If the pod template includes a volume, which refers to a specific PersistentVolumeClaim, all replicas of the ReplicaSet will use the exact same PersistentVolumeClaim and therefore the same PersistentVolume bound by the claim. We will also define one endpoint (service) for each pod inside of our ReplicaSets, since each pod replica is an exact copy of the other.

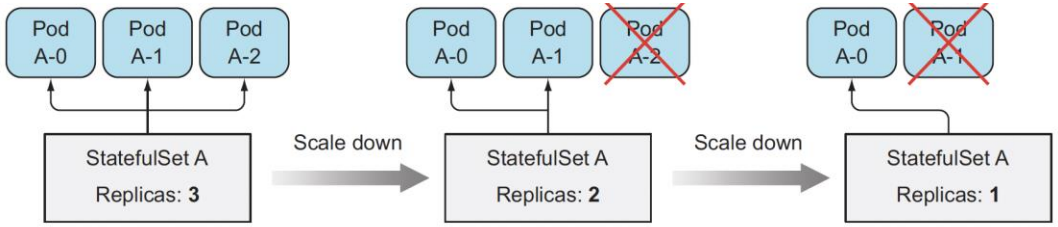
What if we needed to scale-out pods that each needed separate storage, and separate endpoints (service)? We could try to have a ReplicaSet for each such pod, in that case. However, what if we have 1000s of such pods?

VERITAS

StatefulSets



Pods created by a StatefulSet have predictable names (and hostnames), unlike those created by a ReplicaSet



Scaling down a StatefulSet always removes the pod with the highest ordinal index first.

Instead of using a ReplicaSet to run these types of pods, you create a StatefulSet resource, which is specifically tailored to applications where instances of the application must be treated as unique individuals, with each one having a stable name and state.

Each StatefulSet also gets a corresponding governing headless service through which each pod gets its own DNS entry, so its peers and possibly other clients in the cluster can address the pod by its hostname.

A StatefulSet also replaces a lost pod with a new one with the same identity (podname and DNS entry) as before.

Scaling up a StatefulSet creates a new pod instance with the next unused ordinal index. While scaling down, a StatefulSet always removes instances with the highest ordinal index first. Also, StatefulSets scale down only one pod instance at a time unlike ReplicaSets that scale down parallelly (all-at-once).

How to create StatefulSets

Headless service to be used in the StatefulSet: kubia-service-headless.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: kubia
spec:
  clusterIP: None
selector:
  app: kubia
ports:
- name: http
  port: 80
```

Name of the Service

The StatefulSet's governing Service must be headless.

All pods with the app=kubia label belong to this service.

StatefulSet manifest: kubia-statefulset.yaml

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: kubia
spec:
  serviceName: kubia
  replicas: 2
  template:
    metadata:
      labels:
        app: kubia
    spec:
      containers:
      - name: kubia
        image: luksa/kubia-pet
        ports:
        - name: http
          containerPort: 8080
        volumeMounts:
        - name: data
          mountPath: /var/data
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      resources:
        requests:
          storage: 1Mi
      accessModes:
      - ReadWriteOnce
```

Pods created by the StatefulSet will have the app=kubia label.

The container inside the pod will mount the pvc volume at this path.

The PersistentVolumeClaims will be created from this template.

VERITAS

References & Further reading

Books continued:

- Production-ready Microservices, Building standardized systems across an Engineering Organization – O'Reilly.
- Microservices in Action – Manning Publications.
- Kubernetes native Microservices – Manning Publications.

VERITAS