



Persistent Data and Storage



Welcome to the 'Persistent Data and Storage' module. In this module you'll learn about the different types of Kubernetes storage abstractions. You'll learn about StatefulSets and how to use them to manage ordered deployments of Pods and storage. You'll also learn how ConfigMaps can save you time during application deployment by decoupling configuration artifacts from container definitions. Finally, you'll learn how to keep sensitive information safer from accidental exposure using Kubernetes Secrets.

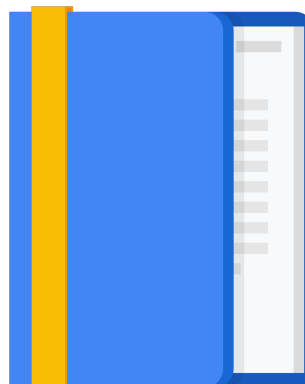
Learn how to ...

Understand and work with Kubernetes storage abstractions.

Run and maintain sets of Pods using StatefulSets.

Use ConfigMaps to decouple configuration from Pods.

Manage and store sensitive authorization and authentication data.



In this module, you'll learn how to understand and work with Kubernetes storage abstractions, which will allow you to select and configure the appropriate storage type for your applications; run and maintain sets of Pods with stable identities and consistent storage, suitable for stateful applications, using StatefulSets; use ConfigMaps to decouple configuration from Pods, so that you don't have to configure variables in individual Pods' specifications, which simplifies management and prevents configuration drift; and manage and store sensitive authorization and authentication configuration data for Pods such as passwords, OAuth tokens, and ssh keys.

Agenda

Volumes

StatefulSets

Lab: Configuring Persistent
Storage for Google Kubernetes
Engine

ConfigMaps

Secrets

Lab: Working with Google
Kubernetes Engine Secrets and
ConfigMaps

Summary

Let's start by introducing Volumes. In this class you'll learn about the types of storage abstractions that Kubernetes provides, such as Volumes and PersistentVolumes. You'll learn about how these differ, and how they can be used to store and share information between Pods.

Kubernetes offers storage abstraction options

Volumes

Are a directory which is accessible to all of the containers in a Pod.

Some Volumes are ephemeral.

Some Volumes are persistent.

PersistentVolumes

Manage durable storage in a cluster.

Are independent of the Pod's lifecycle.

Provisioned dynamically through PersistentVolumeClaims or explicitly created by a cluster admin.

Remember that Kubernetes uses objects to represent the resources it manages. This rule applies to storage as well as to Pods. All these objects function as useful abstractions, which means that you can manage the resources they represent without laborious attention to implementation details. Kubernetes provides storage abstractions as Volumes and PersistentVolumes.

A volume is a directory which is accessible to all of the containers in a Pod. Some Volumes are ephemeral, which means they last only as long as the Pod to which they are attached. You will see examples of these types in this lesson, such as ConfigMap and emptyDir. And some Volumes are persistent, which means that they can outlive a Pod. Regardless of type, Volumes are attached to Pods, not containers. If a Pod isn't mapped to a node any more, the Volume isn't either.

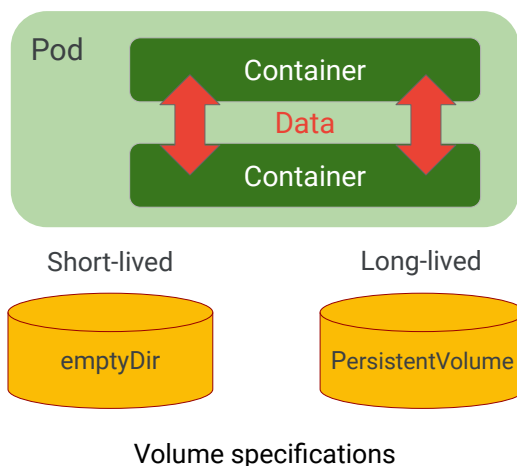
PersistentVolume resources are used to manage durable storage in a cluster. In GKE, a PersistentVolume is typically backed by a persistent disk. You can also use other storage solutions like NFS. Filestore is a NFS solution on Google Cloud. Unlike volumes, the PersistentVolume lifecycle is managed by Kubernetes.

PersistentVolume resources are cluster resources that exist independently of Pods. This means that the disk and data represented by a PersistentVolume continue to exist as the cluster changes and as Pods are deleted and recreated.

PersistentVolume resources can be provisioned dynamically through

PersistentVolumeClaims, or they can be explicitly created by a cluster administrator
You do not have to manually create and delete the backing storage.

Volumes allow containers within a Pod to share data



Remember that you can run multiple containers in a single Pod. You might need to share files between containers, and you might need that storage to exist even after the Pod has terminated.

Volumes allow containers within a Pod to share data and are necessary for Pods to be stateful. Production applications usually need to save state somewhere. So mastering the use of Volumes in Kubernetes is a valuable skill.

Kubernetes offers a variety of Volume types, some which are short lived, Ephemeral Volumes, and some which are long lived, durable volumes. Let's start by taking a look at Ephemeral Volume types.

Ephemeral volume types explained



Ephemeral: shares Pod's lifecycle.



Object can be referenced in a volume.



Stores sensitive info, such as passwords.



Makes data about Pods data available to containers.

- An emptyDir volume is simply an empty directory that allows the containers within the Pod to read and write to and from it. It's created when a Pod is assigned to a node, and it exists as long as the Pod exists. However, it'll be deleted if the Pod is removed from a node for any reason. So don't use emptyDir volumes for data of lasting value. Applications usually use emptyDir for short-term purposes.

Kubernetes creates emptyDir volumes from the node's local disk, or by using a memory-backed file system.

- The configMap resource provides a way to inject application configuration data into Pods from Kubernetes. The data stored in a ConfigMap object can be referenced in a volume, as if it were a tree of files and directories. Your applications can then consume the data. For instance, if you were using a Web server in a Pod, you might use a configMap to set that Web server's parameters.
- Secrets are similar to ConfigMaps. You should use Secrets to store sensitive information, such as passwords, tokens, and ssh keys. Just like ConfigMap, a Secret Volume is created to pass sensitive information to the Pods. These Secret Volumes are backed by in-memory file systems, so the Secrets are

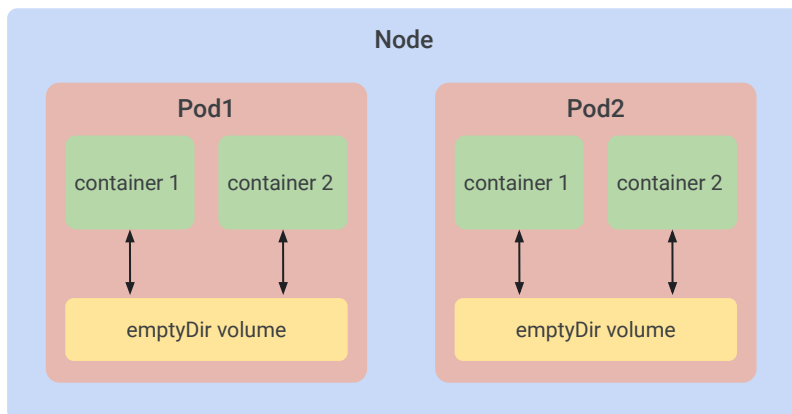
- never written to non-volatile storage. And it's a common practice to obfuscate the values that go into secrets using the familiar base64 encoding.

Beyond that, though, you should not assume that Secrets are secret just because of the way they are configured. Having a differentiation between ConfigMaps and Secrets allows you to manage non-sensitive and sensitive Pod configuration data differently. You will probably apply different permissions to each.

- The downwardAPI Volume type is used to make downwardAPI data available to applications. And what's the downward API? It's a way containers can learn about their Pod environment. For example, suppose your containerized application needs to construct an identifier that's guaranteed to be unique in the cluster. Wouldn't it be easiest to base that identifier on the name of this Pod, since Pod names are unique in the cluster too? The downwardAPI is how your application can fetch the name of the Pod it's running on, if you choose to make it available.

All these volume types are fundamentally similar. ConfigMap, Secret, and downwardAPI are essentially EmptyDir Volumes pre-configured to contain configurations from the GKE API. ConfigMap and Secret are discussed later in this module, while downwardAPI is out of this course's scope.

An emptyDir volume is first created when a Pod is assigned to a node



An emptyDir volume is first created when a Pod is assigned to a node, and exists as long as that Pod is running on that node. The emptyDir volume is initially empty. All containers in the Pod can read and write the same files in the emptyDir volume, though that volume can be mounted at the same or different paths in each container. When a Pod is removed from a node for any reason, the data in the emptyDir is deleted permanently.

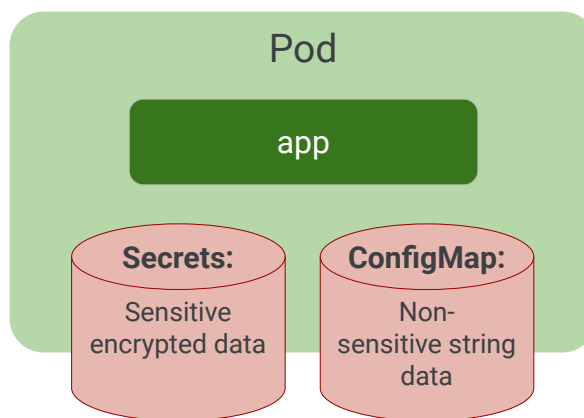
It should be noted that a container crashing does not remove a Pod from a node. The data in an emptyDir volume is safe across container crashes.

Creating a Pod with an emptyDir volume

```
apiVersion: v1
kind: Pod
metadata:
  name: web
spec:
  containers:
    - name: web
      image: nginx
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

Here, we're creating a Pod with an emptyDir volume. Some uses for an emptyDir are a scratch space, such as for a disk-based merge sort, checkpointing a long computation for recovery from crashes or holding files that a content-manager container fetches while a web server container serves the data.

Secret and ConfigMap Volumes are ephemeral



Some Volume types, like Secrets and ConfigMaps, are coupled to the life of the Pod and cease to exist when the Pod ceases to exist.

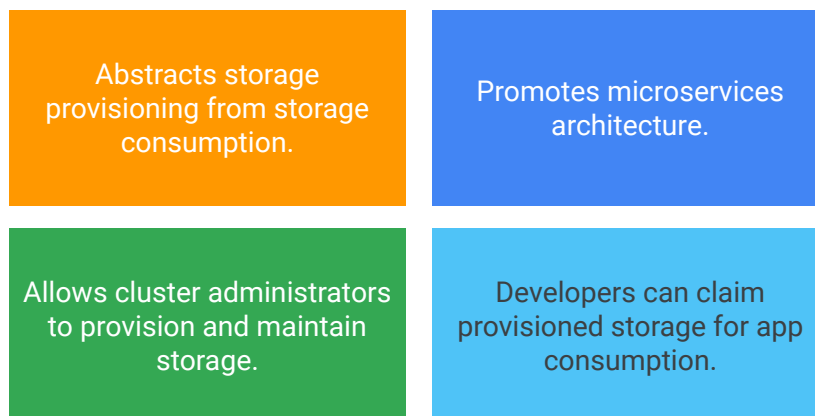
Secrets are a way to store sensitive, encrypted data, such as passwords or keys. These are used to keep you from having to bake sensitive information into your Pods and containers. Secrets are stored in a temporary file system so that they're never written into non-volatile storage.

ConfigMaps are subtly different. They're used for non-sensitive string data; the setting of command-line variables and the storing of configuration and environment variables are natural use cases for ConfigMaps.

While the Secret and ConfigMap Volumes that attach to individual Pods are ephemeral, the objects are not.

All of the examples we've looked at so far are Volumes, and whether the data is long-lived or not, Volumes are tied to the lifecycle of the Pods where they're defined. A PersistentVolume, in contrast, has a lifecycle of its own, independent of the Pod.

The benefits of PersistentVolumes



Kubernetes PersistentVolume objects abstract storage provisioning from storage consumption.

Remember, Kubernetes enables microservices architecture where an application is decoupled into components that can be scaled easily. Persistent storage makes it possible to deal with failures and allow for dynamic rescheduling of components without loss of data. However, should application developers be responsible for creating and maintaining separate Volumes for their application components? Also, how can developers test applications before deploying into production without modifying the Pod manifests for their applications? Whenever you have to reconfigure things to go from test to production, there's a risk of error. Kubernetes' PersistentVolume abstraction resolves both of these issues.

Using PersistentVolumes, a cluster administrator can provision a variety of Volume types. The cluster administrator can simply provision storage and not worry about its consumption.

And application developers can easily claim and use provisioned storage using PersistentVolumeClaims without creating and maintaining storage volumes directly. Notice the separation of roles? It's the job of administrators to make persistent volumes available, and the job of developers to use those volumes in applications.

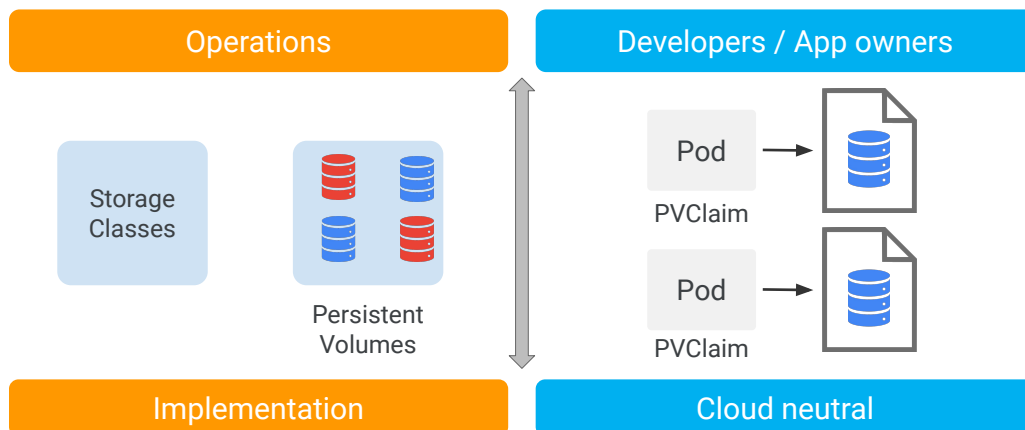
The two job roles can work independently of each other.

When application developers use PersistentVolumeClaims, they don't need to worry about where the application is running: provisioned storage capacity can be claimed by the application regardless of whether it is running on a local site, Google Cloud, or any other cloud provider.

Let's look at what is required for an application owner to consume Compute Engine persistent disk storage, first using Pod-level Volumes and then using Cluster-level PersistentVolumes and PersistentVolumeClaims in Pod manifests. You will see that the second way is more manageable.

In GKE the default StorageClass is configured to dynamically provision gcePersistentDisk based PersistentVolumes by default so PersistentVolumeClaims can be used for standard persistent volumes without any additional preparation.

PersistentVolumeClaims and PersistentVolumes separate storage consumption from provisioning



In order to use PersistentVolumes the operations team that owns the specific cloud implementation define the storage classes and manage the actual implementation details of the Persistent Volumes.

The developers and application owners use PersistentVolumeClaims to request the quantity of storage and storage class, which determines the type of storage.

This allows the operations team to manage the cloud services they wish to use and allows the application owners to focus on what their application requires rather than the specific implementation detail.

Creating a Compute Engine persistent disk using a gcloud command

```
$ gcloud compute disks create  
--size=100GB  
--zone=us-central1-a demo-disk
```

Google Cloud's Compute Engine service uses Persistent Disks for virtual machines' disks, and Kubernetes Engine uses the same technology for PersistentVolumes. Persistent Disks are network-based block storage that can provide durable storage. First a 100-GB Compute Engine Persistent Disk is created using a gcloud command. Before this Persistent Disk can be used by any Pod, someone or some process must create it, and that person or process must have Compute Engine administration rights.

The old but no longer best way of attaching a Pod to a Persistent Disk

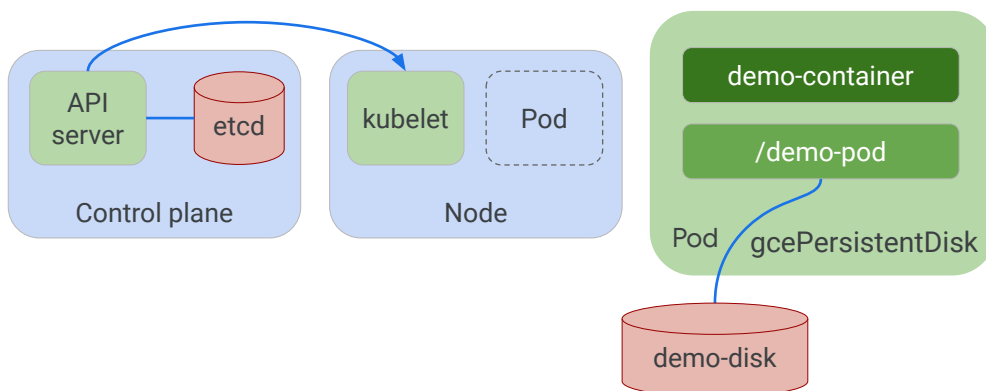
```
[...]
spec:
  containers:
  - name: demo-container
    image: gcr.io/hello-app:1.0
    volumeMounts:
    - mountPath: /demo-pod
      name: pd-volume
  volumes:
  - name: pd-volume
    gcePersistentDisk:
      pdName: demo-disk
      fsType: ext4
```

Google Cloud

As we've seen earlier with the NFS Volume example, this Compute Engine Persistent Disk can be attached to a Pod by specifying a `gcePersistentDisk` Volume type inside the Pod manifest.

In this Pod manifest, the Compute Engine Persistent Disk Volume is described in the `gcePersistentDisk` field as `pdName: demo-disk`, and file system type: `ext4`. The `pdName` must correspond to a Compute Engine Persistent Disk that has already been created. This is the old way of attaching to Persistent Disks, and it's no longer the best way to do it. I'm showing it to you here because you might see it in existing applications.

Creating a Compute Engine Persistent Disk



When the Pod is created, Kubernetes uses the Compute Engine API to attach the Persistent Disk to the node on which the Pod is running. The Volume is automatically formatted and mounted to the container. If this Pod is moved to another node, Kubernetes automatically detaches the Persistent Disk from the existing node and re-attaches it to the newer node.

Configuring Volumes in Pods makes portability difficult

```
Volumes:
  pd-volume:
    Type:          GCEPersistentDisk
    PDName:        demo-disk
    FSType:        ext4
    Partition:     0
```

```
Volumes:
  pd-volume:
    Type:          vsphereVolume
    PDName:        demo-disk
    FSType:        ext4
    Partition:     0
```

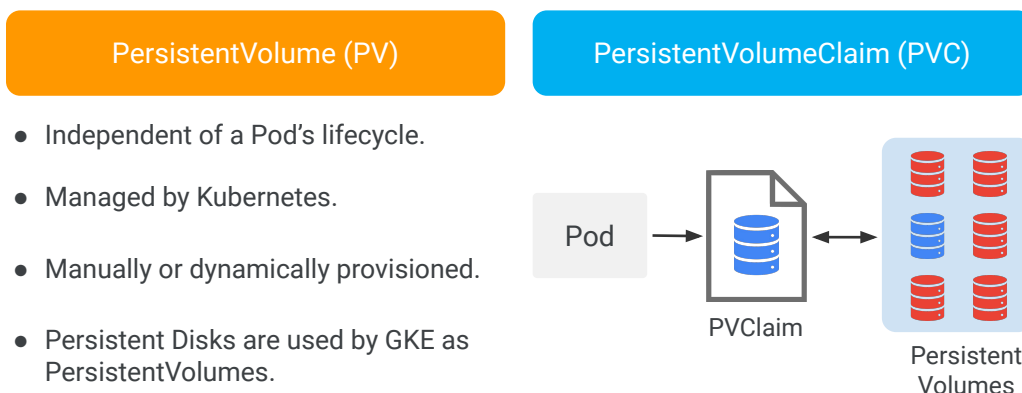
Google Cloud

You can confirm that a Volume was mounted successfully using the `kubectl describe Pod` command. Don't forget that a Persistent Disk must be created before it can be used. The Persistent Disk can have pre-populated data that can be shared by the containers within a Pod. That's very convenient. However, the application owner must include the specific details of the Persistent Disk inside the Pod manifest for their application and must confirm that the Persistent Disk already exists. That's not very convenient.

Hard coding Volume configuration information into Pod specifications in this way means you may have difficulty porting data from one cloud to another. On GKE, Volumes are usually configured to use Compute Engine Persistent Disks. In your on-premises Kubernetes cluster, they might be VMware vSphere volume files, for example, or even physical hard drives.

Whenever you need to reconfigure an application to move it from one environment to another, there's a risk of error. To address this problem, Kubernetes provides an abstraction called Persistent Volumes. This abstraction lets a Pod claim a Volume of a certain size, or of a certain name, from a pool of storage without forcing you to define the storage type details inside the Pod specification.

PersistentVolumes abstraction has two components



Google Cloud

Let's take a closer look at how PersistentVolumes make the use of network storage like this more manageable. The PersistentVolume abstraction has two components: PersistentVolume (PV) and PersistentVolumeClaim (PVC).

PersistentVolumes are durable and persistent storage resources managed at the cluster level. Although these cluster resources are independent of the Pod's lifecycle, a Pod can use these resources during its lifecycle. However, if a Pod is deleted, a PersistentVolume and its data continue to exist.

These Volumes are managed by Kubernetes and can be manually or dynamically provisioned.

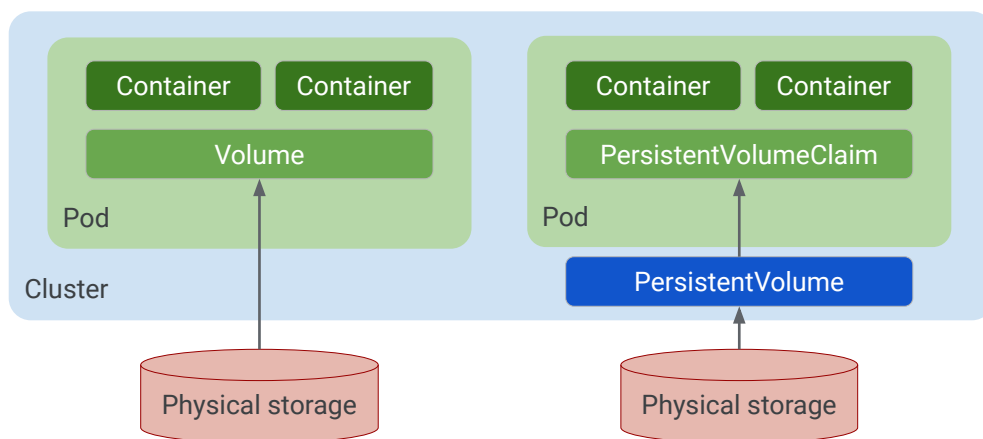
GKE can use Compute Engine Persistent Disks as PersistentVolumes.

PersistentVolumeClaims are requests and claims made by Pods to use PersistentVolumes. Within a PersistentVolumeClaim object, you define a Volume size, access mode, and StorageClass. What's a StorageClass? It's a set of storage characteristics that you've given a name to.

A Pod uses this PersistentVolumeClaim to request a PersistentVolume. If a PersistentVolume matches all the requirements defined in a PersistentVolumeClaim,

the PersistentVolumeClaim is bound to that PersistentVolume. Now, a Pod can consume storage from this PersistentVolume.

PersistentVolumes must be claimed



What's the critical difference between using Pod-level Volumes and Cluster-level PersistentVolumes for storage? PersistentVolumes provide a level of abstraction that lets you decouple storage administration from application configuration. The storage in a PersistentVolume must be bound with a PersistentVolumeClaim in order to be accessed by a Pod.

Pod manifest with a Volume

```
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
spec:
  containers:
  - name: demo-container
    image: gcr.io/hello-app:1.0
    volumeMounts:
    - mountPath: /demo-pod
      name: pd-volume
  volumes:
  - name: pd-volume
    gcePersistentDisk:
      pdName: demo-disk
      fsType: ext4
```

Earlier, you saw this example of how to specify a Compute Engine Persistent Disk Volume in this sample Pod manifest.

Creating a PersistentVolume manifest

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pd-volume
spec:
  storageClassName: "standard"
  capacity:
    storage: 100G
  accessModes:
    - ReadWriteOnce:
  gcePersistentDisk:
    pdName: demo-disk
    fsType: ext4
```

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

PVC StorageClassName
must match the
PV StorageClassName

Google Cloud

Here's how you create a PersistentVolume manifest for the same storage. Let's take a closer look at how this is used to make managing storage configuration for Pods easier.

First, you specify the Volume capacity.

Then the storageClassName. StorageClass is a resource used to implement PersistentVolumes. Note that the PVC uses the StorageClassName when you define the PVC in a Pod, and it must match the PV StorageClassName for the claim to succeed.

GKE has a default StorageClass named 'standard' to use the Compute Engine Standard Persistent Disk type, as shown here on the right. In this example, the PV definition on the left matches the GKE default StorageClass. In GKE clusters, a PVC with no defined StorageClass will use this default StorageClass and provide storage using a standard Persistent Disk.

Create a new StorageClass to use an SSD Persistent Disk

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pd-volume
spec:
  storageClassName: "ssd"
  capacity:
    storage: 100G
  accessModes:
    - ReadWriteOnce:
  gcePersistentDisk:
    pdName: demo-disk
    fsType: ext4
```

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

If you want to use an SSD Persistent Disk, you can create a new StorageClass, such as this example named `ssd`. A PVC that uses this new StorageClass named `ssd` will only use a PV that also has a StorageClass named `ssd`. In this instance, an SSD Persistent Disk will be used.

Viewing the new storage class in the Cloud Console

<input type="checkbox"/>	Name ↑	Provisioner	Type	Zone	Cluster
<input type="checkbox"/>	premium-rwo	pd.csi.storage.gke.io	pd-ssd		standard-cluster-1
<input type="checkbox"/>	standard	kubernetes.io/gce-pd	pd-standard		standard-cluster-1
<input type="checkbox"/>	standard-rwo	pd.csi.storage.gke.io	pd-balanced		standard-cluster-1

Once you create the new storage class with a `kubectl apply` command, you can view it in the Cloud Console.

By the way, don't confuse Kubernetes StorageClasses with Storage Classes that Google Cloud Storage makes available. Although the features have the same name, they are unrelated, because they come from different services and govern different features. Google Cloud Storage is object storage for the web, while Kubernetes StorageClasses are choices for how PersistentVolumes are backed.

The AccessModes you specify determine how this Volume can be read from or written to

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pd-volume
spec:
  storageClassName: "standard"
  capacity:
    storage: 100G
  accessModes:
    - ReadWriteOnce
  gcePersistentDisk:
    pdName: demo-disk
    fsType: ext4
```

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pd-volume
spec:
  storageClassName: "standard"
  capacity:
    storage: 100G
  accessModes:
    - ReadOnlyMany
  gcePersistentDisk:
    pdName: demo-disk
    fsType: ext4
```

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pd-volume
spec:
  storageClassName: "nfs"
  capacity:
    storage: 100G
  accessModes:
    - ReadWriteMany
  nfs:
    path: /tmp
    server: 172.17.0.2
```

The types of AccessModes that are available depend on the volume type. As we discuss the various AccessModes, we will mention what types of volumes offer them by looking at these examples of persistent volume definitions.

- ReadWriteOnce mounts the Volume as read-write to a single node.
- ReadOnlyMany mounts a Volume as read-only to many nodes.
- ReadWriteMany mounts Volumes as read-write to many nodes.

For most applications, Persistent Disks are mounted as ReadWriteOnce. They can also be mounted as ReadOnlyMany when the data is static; but Google Cloud Persistent Disks do not support ReadWriteMany. Some other network volume types, such as NFS, do support the ReadWriteMany accessMode.

You can create a Persistent Volume from a YAML manifest

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pd-volume
spec:
  storageClassName: "standard"
  capacity:
    storage: 100G
  accessModes:
    - ReadWriteOnce:
  gcePersistentDisk:
    pdName: demo-disk
    fsType: ext4
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pd-volume-claim
spec:
  storageClassName: "standard"
  accessModes:
    - ReadWriteOnce:
  resources:
    requests:
      storage: 100G
```

Google Cloud

You can create a Persistent Volume from a YAML manifest using a `kubectl apply` command. This PV example shown here named `pd-volume` will have 100 GB of storage allocated and will allow `ReadWriteOnce` access. It will use a standard Persistent Disk based on the `storageClassName`. This PV would be maintained by cluster administrators. For example, among the administrator's responsibilities would be backing it up. You do backups, right? Of course you do. You can back up a Persistent Disk, regardless of whether it is in use by Compute Engine or Kubernetes Engine, by creating snapshots of it.

Persistent Volumes can't be added to Pod specifications directly. Instead, you must use `PersistentVolumeClaims`. In order for this `PersistentVolumeClaim` to claim the `PersistentVolume`, their `storageClassNames` and `accessModes` must match. Also, the amount of storage requested in a `PersistentVolumeClaim` must be within a `PersistentVolume`'s storage capacity. Otherwise, the claim will fail. Notice that this claim wants 100 gigabytes, and the `PersistentVolume` offers 100 gigabytes, so it will succeed, assuming that the `PersistentVolume` is healthy.

The traditional way to provide durable or persistent storage for a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
spec:
  containers:
    - name: demo-container
      image: gcr.io/hello-app:1.0
      volumeMounts:
        - mountPath: /demo-pod
          name: pd-volume
  volumes:
    - name: pd-volume
      gcePersistentDisk:
        pdName: demo-disk
        fsType: ext4
```

So let's be sure you understand how PersistentVolumes and PersistentVolumeClaims are used. Earlier you saw the traditional way to provide that durable or persistent storage for a Pod: by adding an appropriate Volume specification to a Pod manifest, as shown again here.

The modern, easier-to-manage way is to use the PersistentVolume abstraction

```
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
spec:
  containers:
    - name: demo-container
      image: gcr.io/hello-app:1.0
      volumeMounts:
        - mountPath: /demo-pod
          name: pd-volume
  volumes:
    - name: pd-volume
      PersistentVolumeClaim:
        claimName: pd-volume-claim
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pd-volume-claim
spec:
  storageClassName: "standard"
  accessModes:
    - ReadWriteOnce:
  resources:
    requests:
      storage: 100G
```

Add a PersistentVolumeClaim to the Pod, as shown here on the left. In our example, the PersistentVolumeClaim named 'pd-volume-claim' has the 'standard' storageClassName, the 'ReadWriteOnce' accessMode, and a requested capacity of 100 gigabytes. When this Pod is started, GKE will look for a matching PV with the same storageClassName and accessModes and sufficient capacity. The specific cloud implementation doesn't really matter, the specific storage used to deliver this storage class is something the cluster administrators, not the application developers, control.

What could go wrong with this? Well, what if application developers claim more storage than has already been allocated to PersistentVolumes? PersistentVolumes are managed by cluster administrators, but application developers make the PersistentVolumeClaims, and this could lead to storage allocation failures.

An alternative option is Dynamic Provisioning

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pd-volume-claim
spec:
  storageClassName: "standard"
  accessModes:
    - ReadWriteOnce:
  resources:
    requests:
      storage: 100G
```

So what happens if there isn't an existing PersistentVolume to satisfy the PersistentVolumeClaim? If it cannot find an existing PersistentVolume, Kubernetes will try to provision a new one dynamically.

By default, Kubernetes will try to dynamically provision a PersistentVolume if the PersistentVolumeClaim's storageClassName is defined and an appropriate PV does not already exist. If a matching PersistentVolume already exists, as seen in our previous example, Kubernetes will bind it to the claim.

If you omit storageClassName, the PersistentVolumeClaim will use the default StorageClass, which, in GKE, is named 'standard.' That's a standard Persistent Disk. Dynamic provisioning will only work in this case if it is enabled on the cluster.

All of this is handled by GKE. The application owner does not have to ensure that the underlying storage has been provisioned for them to use, and does not need to embed the details of the underlying storage into the Pod manifest.

The PersistentVolume can be retained when the PersistentVolumeClaim is deleted

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pd-volume-claim
spec:
  storageClassName: "standard"
  accessModes:
    - ReadWriteOnce:
  resources:
    requests:
      storage: 100G
  persistentVolumeReclaimPolicy: Retain
```

By default, deleting the PersistentVolumeClaim will also delete the provisioned PersistentVolume. If you want to retain the PersistentVolume, set its `persistentVolumeReclaimPolicy` to 'Retain'. In general, PersistentVolumeClaims should be deleted when their underlying PersistentVolume is no longer required.

Regional Persistent Disks improves availability

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: ssd
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
  replication-type: regional-pd
  zones: us-central1-a, us-central1-b
```

Google Cloud

What if you want to ensure high availability for your PersistentVolumes? You can improve the availability of Compute Engine Persistent Disks by deploying them as regional persistent disks. Regional persistent disks replicate data between two zones in the same region, which improves availability. A regional persistent disk can be launched manually or dynamically and provisioned by configuring additional fields in a storageclass.

If a zonal outage occurs, Kubernetes can failover those workloads that use the volume to the other zone. You can use regional persistent disks to build highly available solutions for stateful workloads on GKE.

To use regional persistent disks in Kubernetes, create a StorageClass that specifies the 'regional-pd' replication-type in its definition. Then use that StorageClass when you define Persistent Volumes.

At the time this class was written, Compute Engine regional persistent disks were limited to spanning two zones as shown in the yaml here.

Agenda

Volumes

StatefulSets

Lab: Configuring Persistent Storage for Google Kubernetes Engine

ConfigMaps

Secrets

Lab: Working with Google Kubernetes Engine Secrets and ConfigMaps

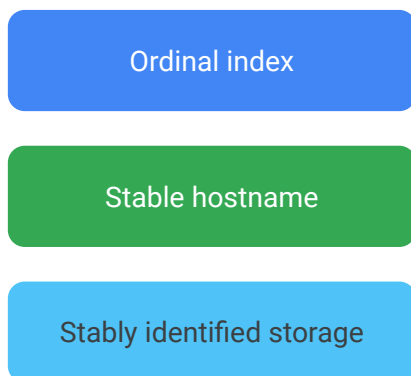
Summary

In the previous section, we explained that Persistent Volumes provide durable storage for a Pod. You can also use Persistent Volumes for other controllers, such as Deployments and StatefulSets. Remember, a Deployment is simply a Pod template that runs and maintains a set of identical Pods, commonly known as *replicas*. You can use these Deployments for stateless applications. Deployment replicas can share an existing Persistent Volume using a `ReadOnlyMany` or `ReadWriteMany` access mode. `ReadWriteMany` access mode can only be used for storage types that support it, such as NFS systems.

The other access mode you learned about, `ReadWriteOnce`, isn't recommended for Deployments, because the replicas need to attach and reattach to Persistent Volumes dynamically. If a first Pod needs to detach itself, the second Pod needs to be attached first; however, the second Pod cannot attach because the first Pod is already attached. This creates a deadlock. So neither Pod can make progress. StatefulSets resolve this deadlock. Whenever your application needs to maintain state in Persistent Volumes, managing it with a StatefulSet rather than a Deployment is the way to go.

StatefulSets are useful for stateful applications. StatefulSets run and maintain a set of Pods, just like Deployments do. A StatefulSet object defines a desired state, and its controller achieves it. However, unlike Deployments, StatefulSets maintain a persistent identity for each Pod.

Each Pod in a StatefulSet maintains a persistent identity



Each Pod in a StatefulSet maintains a persistent identity and has an ordinal index with a relevant Pod name, a stable hostname, and stably identified persistent storage that is linked to the ordinal index. What's an ordinal index? It's just a unique sequential number that is assigned to each Pod in the stateful set. This number defines the pod's position in the set's sequence of Pods.

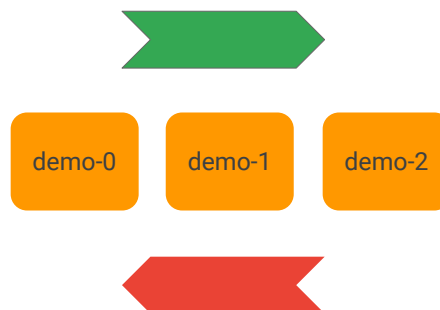
Deployment, scaling, and updates are ordered using the ordinal index of the Pods within a StatefulSet

StatefulSet named *demo* launches 3 replicas.

- Launches Pods sequentially named.
- Scaling and rolling updates happen in reverse order.

Pod Management Policy

- OrderedReady (Default)
- Parallel



Deployment, scaling, and updates are ordered using the ordinal index of the Pods within a StatefulSet. For example, if a StatefulSet named 'demo' launches 3 replicas, it will launch Pods named demo-0, demo-1, and demo-2 sequentially. This means that all of its predecessors must be Running and Ready before an action is taken on a newer Pod. For example, if demo-0 is not Running and Ready, demo-1 will not be launched. If demo-0 fails after demo-1 is Running and Ready, but before the creation of demo-2, demo-2 will not be launched until demo-0 is relaunched and becomes Running and Ready.

Scaling and rolling updates happen in reverse order, which means demo-2 would be changed first.

This depends on the Pod management policy being set to the default – OrderedReady state. If you want to launch Pods in parallel, without waiting for the Pods to maintain Running and Ready state, change the PodManagementPolicy to Parallel.

Characteristics of StatefulSets

- StatefulSets are designed for stateful applications.
- StatefulSets use volumeClaimTemplates to launch each Pod's unique PVCs.
- PVCs can use ReadWriteOnce access mode.

As the name suggests, StatefulSets are useful for stateful applications.

With stable storage, StatefulSets use a unique Persistent Volume Claim for each Pod. So that each Pod can maintain its own individual state, it must have reliable long-term storage to which no other Pod writes.

These PersistentVolumeClaims use ReadWriteOnce access mode for applications.

StatefulSets require a service to control networking

```
apiVersion: v1
kind: Service
metadata:
  name: demo-service
  labels:
    app: demo
spec:
  ports:
    - port: 80
      name: web
  clusterIP: None
  selector:
    app: demo
---
```

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: demo-statefulset
spec:
  selector:
    matchLabels:
      app: demo
  serviceName: demo-service
  replicas: 3
  updateStrategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: demo
```

First, StatefulSets require a service to control their networking. Sometimes you may not want or need load-balancing and a single service IP. In this case, you create a headless service by specifying “None” for the cluster IP in the Service definition, as shown here on the left.

The StatefulSet defined on the right refers to this Service using the serviceName field.

A label selector is required for the Service

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: demo-statefulset
spec:
  selector:
    matchLabels:
      app: demo
  serviceName: demo-service
  replicas: 3
  updateStrategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: demo
```

```
[...]
spec:
  containers:
    - name: demo-container
      image: k8s.gcr.io/demo:0.1
      [...]
  volumeClaimTemplates:
    - metadata:
        name: demo-pvc
      spec:
        accessModes: ["ReadWriteOnce" ]
        resources:
          requests:
            storage: 1Gi
```

Second, a label selector is required for the Service, and this must match the template's labels defined in the template section of the StatefulSet definition.

Define the container details

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: demo-statefulset
spec:
  selector:
    matchLabels:
      app: demo
  serviceName: demo-service
  replicas: 3
  updateStrategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: demo
```

```
[...]
spec:
  containers:
    - name: demo-container
      image: k8s.gcr.io/demo:0.1
      ports:
        - containerPort: 80
          name: web
      volumeMounts:
        - name: www
          mountPath: /usr/share/web
[...]
```

```
volumeClaimTemplates:
- metadata:
```

Third, you define the container details, including the image, containerPort for the Service, and Volume mounts.

Specify volumeClaimTemplates

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: demo-statefulset
spec:
  selector:
    matchLabels:
      app: demo
  serviceName: demo-service
  replicas: 3
  updateStrategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: demo
```

```
[...]
spec:
  containers:
    - name: demo-container
      image: k8s.gcr.io/demo:0.1
      [...]
  volumeClaimTemplates:
    - metadata:
        name: demo-pvc
      spec:
        accessModes: ["ReadWriteOnce" ]
        resources:
          requests:
            storage: 1Gi
```

Most importantly, volumeClaimTemplates are specified under the template section. Here, the volumeClaimTemplate is named, and the spec is the same as the PersistentVolumeClaim that is required by the Pods in this StatefulSet.

Agenda

Volumes

StatefulSets

Lab: Configuring Persistent
Storage for Google Kubernetes
Engine

ConfigMaps

Secrets

Lab: Working with Google
Kubernetes Engine Secrets and
ConfigMaps

Summary

Lab Intro

Configuring Persistent Storage for
Google Kubernetes Engine



In this lab, you'll set up `PersistentVolumes` and `PersistentVolumeClaims`. `PersistentVolumes` are storage that is available to a Kubernetes cluster. `PersistentVolumeClaims` enable Pods to access `PersistentVolumes`. Without `PersistentVolumeClaims`, Pods are mostly ephemeral, so you should use `PersistentVolumeClaims` for any data that you expect to survive Pod scaling, updating, or migrating.

The tasks that you'll perform include creating manifests for PVs and PVCs for Compute Engine persistent disks, mounting Compute Engine persistent disk PVCs as volumes in Pods, and using manifests to create `StatefulSets`. You'll also mount Compute Engine persistent disk PVCs as Volumes in `StatefulSets` and verify the connection of pods in `StatefulSets` to particular PVs as the Pods are stopped and restarted.

Agenda

Volumes

StatefulSets

Lab: Configuring Persistent
Storage for Google Kubernetes
Engine

ConfigMaps

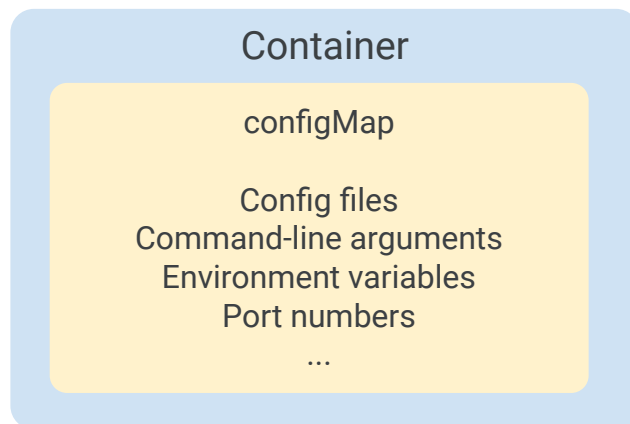
Secrets

Lab: Working with Google
Kubernetes Engine Secrets and
ConfigMaps

Summary

In this class, we'll look at ConfigMaps. ConfigMaps decouple configuration from Pods. This means that you don't have to enter the same information across multiple Pods' specifications. You store this configuration in one place and maintain it as a single source of truth. This prevents configuration drift.

ConfigMaps promote Twelve-Factor-ness



Using configMap, you can store configuration files, command-line arguments, environment variables, port numbers, and other configuration artifacts and make them available inside Containers. This makes your applications more portable and manageable without requiring them to be Kubernetes-aware.

Creating a ConfigMap using a simple kubectl create command

```
$ kubectl create configmap [NAME] [DATA]
```

ConfigMaps can be created from literal values, files, and directories using a simple kubectl create command. These data sources contain key-value pairs. Let's see how each is created.

Creating a ConfigMap using literal values

```
$ kubectl create configmap demo --from-literal=lab.difficulty=easy \
--from-literal=lab.resolution=high
```

kubectl

```
$ kubectl describe configMaps demo
Name:         demo
Namespace:    default
...
Data
====
lab.difficulty:
----
easy
lab.resolution:
----
high
```

Cloud Console

demo	
Cluster	projectdemo
Namespace	default
Created	Oct 22, 2018, 4:43:27 PM
Labels	No labels set
Annotations	Not set
Data	
lab.difficulty	easy
lab.resolution	high

Here a ConfigMap named 'demo' is created using literal values.

Two key-value pairs are defined: lab.difficulty=easy, and lab.resolution=high.

You can add multiple key-value pairs.

You can view the details of configMaps using kubectl or the Cloud Console.

Creating a ConfigMap using files

```
$ kubectl create configmap demo --from-file=demo/color.properties \  
--from-file=demo/ui.properties
```

kubectl

```
Name:      demo  
Namespace: default  
Labels:    <none>  
Annotations: <none>
```

Data

====

```
color.properties: |-  
  color.good=green  
  color.bad=red  
ui.properties: |-  
  resolution=high  
  AA0=enabled
```

Cloud Console

demo

Cluster	projectdemo
Namespace	default
Created	Oct 22, 2018, 5:03:07 PM
Labels	No labels set
Annotations	Not set

Data

color.properties	color.good=green color.bad=red
ui.properties	resolution=high AA0=enabled

Here's another ConfigMap, this time created using the from-file syntax. These files contain multiple key-values. You can add multiple files to a ConfigMap. You could check these files into your source code control system, to maintain their versioning and history.

You can add a key name instead of using source filenames

```
$ kubectl create configmap [NAME] --from-file=[KEY_NAME_1]=[FILE_PATH_1] \  
--from-file=[KEY_NAME_2]=[FILE_PATH_2]
```

```
$ kubectl create configmap demo --from-file=Color=demo/color.properties \  
--from-file=Graphics=demo/ui.properties
```

If you want to specify names for your keys, you can add a key name as an alternative to using the source filenames. The syntax here is very similar to the default from-file syntax in the previous slide, but here an additional key value is inserted to rename the key used for the file called color.properties to the key value Color, and to rename the key used for the file called ui.properties to a key called Graphics.

As always, you can verify the contents of your ConfigMaps this using kubectl or the Cloud Console.

Creating a ConfigMap using a directory

```
$ mkdir -p demo/  
$ wget https://k8s.io/demo/color.properties -O demo/color.properties  
$ wget https://k8s.io/demo/ui.properties -O demo/ui.properties
```

```
$ kubectl create configmap demo  
--from-file=demo/
```

demo

Cluster	projectdemo
Namespace	default
Created	Oct 22, 2018, 5:03:07 PM
Labels	No labels set
Annotations	Not set

Data

color.properties	color.good=green color.bad=red
ui.properties	resolution=high AAO=enabled

Instead of specifying each file from the same directory, you can simply use the directory name directly. All files within a directory are added to the ConfigMap. Here, a directory is created and files are stored in it. When you add a directory to the ConfigMap, the files contained in the directory are also added.

Creating a ConfigMap from a manifest

```
apiVersion: v1
data:
  color.properties: |-
    color.good=green
    color.bad=red
  ui.properties: |-
    resolution=high
    AAO=enabled
kind: ConfigMap
metadata:
  name: demo
```

ConfigMaps can also be created from a manifest. Here, the data is the same as in the previous examples. Simply applying this manifest using a `kubectl apply` will create the ConfigMap.

How should you choose among these ways of defining ConfigMaps? Select whichever way works best for your operations. Remember that the purpose of ConfigMaps is to support separation of concerns by ensuring that configuration data isn't embedded literally in your applications.

Configuring a Pod to use a ConfigMap

- As a container environment variable.
- In Pod commands.
- By creating a Volume.

Pods refer to ConfigMaps in three ways: as a container environment variable, in Pod commands, or by creating a Volume.

Using a ConfigMap as a container environment variable

```
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
spec:
  containers:
  - name: test-container
    image: /busybox
    env:
    - name: VARIABLE_DEMO
      valueFrom:
        configMapKeyRef:
          name: demo
          key: lab.difficulty
```

demo

Cluster	projectdemo
Namespace	default
Created	Oct 22, 2018, 4:43:27 PM
Labels	No labels set
Annotations	Not set

Data

lab.difficulty	easy
lab.resolution	high

Here, a single ConfigMap is used in the Pod as a container environment variable.

Within an env field, a container environment variable is named as VARIABLE_DEMO. The values are retrieved using configMapKeyRef.

Multiple variables can be added from the same or different ConfigMaps.

Using a ConfigMap in Pod commands

```
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
spec:
  containers:
    - name: demo-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "echo ${VARIABLE_DEMO}" ]
      env:
        - name: VARIABLE_DEMO
          valueFrom:
            configMapKeyRef:
              name: demo
              key: lab.difficulty
```

After the container environmental variables are defined, they can be used inside Pod manifest commands using the syntax shown here: you put a dollar sign and an opening parenthesis in front of the environment variable's name, and a closing parenthesis after it. If you are familiar with Linux shells, this may look like an error to you. Linux shells use the same syntax for a different purpose.

Using a ConfigMap by creating a Volume

```
[...]
Kind: Pod
spec:
  containers:
  - name: demo-container
    image: k8s.gcr.io/busybox
    volumeMounts:
    - name: config-volume
      mountPath: /etc/config
  volumes:
  - name: config-volume
    configMap:
      name: demo
```

You can also add ConfigMap data into an ephemeral Volume.

In this example, a Volume named *config-volume* is created in the Volumes section, with a ConfigMap named *demo*.

The result: a ConfigMap Volume is created for this Pod. All the data from the ConfigMap is stored in this ConfigMap Volume as files, and then this Volume is mounted to the container using the mountPath directory.

Each node's Kubelet periodically syncs with ConfigMap to keep the ConfigMap Volume updated



When a ConfigMap Volume is already mounted and the source ConfigMap is changed, the projected keys are eventually updated. What does “eventually” mean here? It’s on the order of seconds or minutes. If you have a piece of configuration data that will change more rapidly than that, you should probably implement a microservice to provide its value to Pods, rather than using a ConfigMap. You cannot force this update to happen.

Agenda

Volumes

StatefulSets

Lab: Configuring Persistent
Storage for Google Kubernetes
Engine

ConfigMaps

Secrets

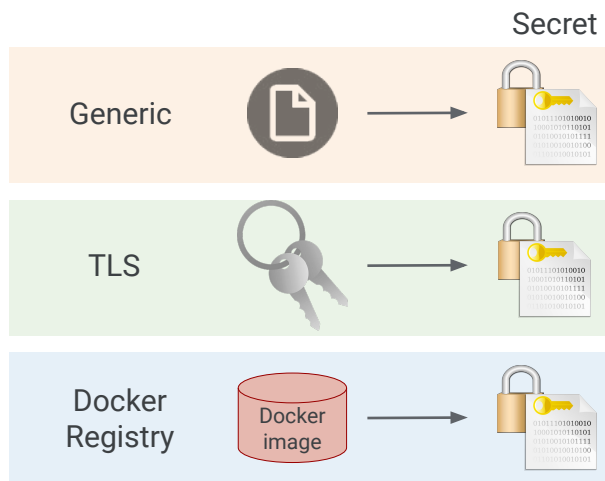
Lab: Working with Google
Kubernetes Engine Secrets and
ConfigMaps

Summary

This final lesson of the module looks at Secrets in more detail. Just like ConfigMap, Secrets pass information to Pods. Secrets are similar to ConfigMaps, but it's a convention that Kubernetes applications use Secrets rather than ConfigMaps to store sensitive information such as passwords, tokens, and SSH keys. Secrets let you manage sensitive information in their own control plane. Secrets also help to ensure Kubernetes doesn't accidentally output this data to logs.

In spite of the name, Kubernetes Secrets are not truly secret. If your application is managing high-value assets, or if you face stringent regulatory requirements, you should consider key management systems like Cloud Key Management Service (Cloud KMS) for full secret management.

There are three types of Secrets



The Generic type is used when creating Secrets from files, directories, or literal values.

The TLS type uses an existing public/private encryption key pair. To create one of these, you must give Kubernetes the public key certificate, encoded in PEM format, and you must also supply the private key of that certificate.

The Docker-Registry Secret type can be used to pass credentials for an image registry to a Kubelet so it can pull a private image from the Docker Registry on behalf of your Pod.

In GKE, Container Registry integrates with Cloud Identity and Access Management, so you may not need to use this Secret type.

In Secrets, you supply values as base-64-encoded strings

```
$ echo -n 'admin' | base64
YWRtaW4=
$ echo -n 'kubernetes' | base64
a3ViZXJhZXRlcw==
```

```
apiVersion: v1
kind: Secret
metadata:
  name: demo-secret
type: Opaque
data:
  username: YWRtaW4=
  password: a3ViZXJhZXRlcw==
```

Just like ConfigMap, generic-type Secrets are stored in key-value pairs. However, in Secrets, you supply values as base-64-encoded strings. Be careful. Base-64 encoding is not encryption. If you need to thoroughly protect data you should use encryption and an encryption key management system like Cloud KMS.

These encoded strings can then be used in the Secret manifest. Applying this manifest using a `kubectl create` command creates a Secret.

Creating a generic Secret

Creating a Secret using literal values

```
$ kubectl create secret generic demo \
  --from-literal user=admin \
  --from-literal password=kubernetes
```

Creating a Secret using files

```
$ kubectl create secret generic demo \
  --from-file=./username.txt \
  --from-file=./password.txt
```

Creating a Secret using naming keys

```
$ kubectl create secret generic demo \
  --from-file=User=./username.txt \
  --from-file=Password=./password.txt
```

Here are examples of generic types. Note that files should have single entries.

All the entries are packaged together, with filenames as keys and entries inside as values.


Keys can also be named.

Listing a Secret

demo-secret

Cluster	projectdemo
Namespace	default
Created	Oct 22, 2018, 9:23:35 PM
Labels	<i>No labels set</i>
Annotations	<i>Not set</i>

Data

 Sensitive data fields are not displayed in the console.

password	••••••••••
username	••••••••••

NAME	TYPE	DATA	AGE
demo-secret	Opaque	2	1m

```
Name:      demo-secret
Namespace: default
Labels:    <none>
Annotations: <none>
```

Type: Opaque

```
Data
====
password: 10 bytes
username:  5 bytes
```

You might have noticed that this is similar to ConfigMap. One difference however, is that encoded strings aren't visible in the Cloud Console, and encoded strings aren't displayed when you use the `kubectl 'get' or 'describe' commands`.

Secrets can be consumed as environment variables in a Pod by its containers

```
[...] kind: Pod
spec:
  containers:
  - name: mycontainer
    image: redis
    env:
    - name: SECRET_USERNAME
      valueFrom:
        secretKeyRef:
          name: demo-secret
          key: username
    - name: SECRET_PASSWORD
      valueFrom:
        secretKeyRef:
          name: demo-secret
          key: password
```

Google Cloud

Here's an example where two environment variables (SECRET_USERNAME and SECRET_PASSWORD) are created. These variables are referenced to a Secret named demo-secret, along with their keys. These variables are automatically decoded.

There are separate control planes (and backing stores) for configmaps and secrets. These provide a mechanism for providers to create a more secure backing store for Secrets that can be used to store and protect your secrets.

GKE has a beta feature that integrates Secrets with Cloud HSM using this technique.

Secrets can also be consumed by creating a Secret Volume in a Pod

```
[...] kind: Pod
spec:
  containers:
  - name: mycontainer
    image: redis
    volumeMounts:
    - name: storagesecrets
      mountPath: "/etc/sup"
      readOnly: true
  volumes:
  - name: storagesecrets
    secret:
      secretName: demo-secret
```

Here, a Secret Volume named *storagesecrets* is created and referred to a Secret named *demo-secret*. This Volume is mounted to the container with read-only access. This Volume can be used by multiple containers within the Pod.

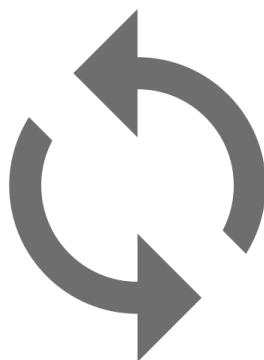
Secret keys can be projected to a specific path

```
[...] kind: Pod
spec:
  containers:
  - name: mycontainer
    image: redis
    volumeMounts:
    - name: storagesecrets
      mountPath: "/etc/sup"
      readOnly: true
  volumes:
  - name: storagesecrets
    secret:
      secretName: demo-secret
      items:
      - key: username
        path: group/users
```

In this case, a password key of Secret will not be projected. If a password key is required, it must also be listed under the items field.

Kubelet periodically syncs with Secrets to keep a Secret Volume updated

Kubelet



Secret

Just like ConfigMap, Kubelet periodically syncs with Secrets to keep a Secret Volume updated. If a Secret that is already attached as a Volume is changed, the keys and values are eventually updated.

Agenda

Volumes

StatefulSets

Lab: Configuring Persistent
Storage for Google Kubernetes
Engine

ConfigMaps

Secrets

Lab: Working with Google
Kubernetes Engine Secrets and
ConfigMaps

Summary

Lab Intro

Working with Google Kubernetes
Engine Secrets and ConfigMaps



In this lab, you'll set up configuration information, both encrypted and unencrypted. Encrypted configuration information is stored as secrets. Unencrypted configuration information is stored as ConfigMaps. This approach avoids hard coding such information into code bases. Credentials, such as API keys, that belong in secrets should never travel inside code repositories like GitHub, unless they're encrypted before going in. Even then though, it's better to keep them separate.

The tasks in the lab will provide you with the opportunity to create secrets and ConfigMaps using the `kubectl` command and manifest files. You'll also consume secrets and ConfigMaps in containers by using environment variables or mounted volumes.

Agenda

Volumes

StatefulSets

Lab: Configuring Persistent
Storage for Google Kubernetes
Engine

ConfigMaps

Secrets

Lab: Working with Google
Kubernetes Engine Secrets and
ConfigMaps

[Summary](#)

Summary

Understand and work with Kubernetes storage abstractions.

Run and maintain sets of Pods using StatefulSets.

Use ConfigMaps to decouple configuration from Pods.

Manage and store sensitive authorization and authentication data.

That concludes Persistent Data and Storage. In this module you learned how to understand and work with Kubernetes storage abstractions which will allow you configure and select the appropriate storage type for your applications; run and maintain sets of Pods with stable identities and consistent storage suitable for stateful applications, using StatefulSets; use ConfigMaps to decouple configuration from Pods, so that you don't have to configure variables in individual Pods' specifications, simplifying management and preventing configuration drift; and manage and store sensitive authorization and authentication configuration data for Pods such as passwords, OAuth tokens, and ssh keys.

