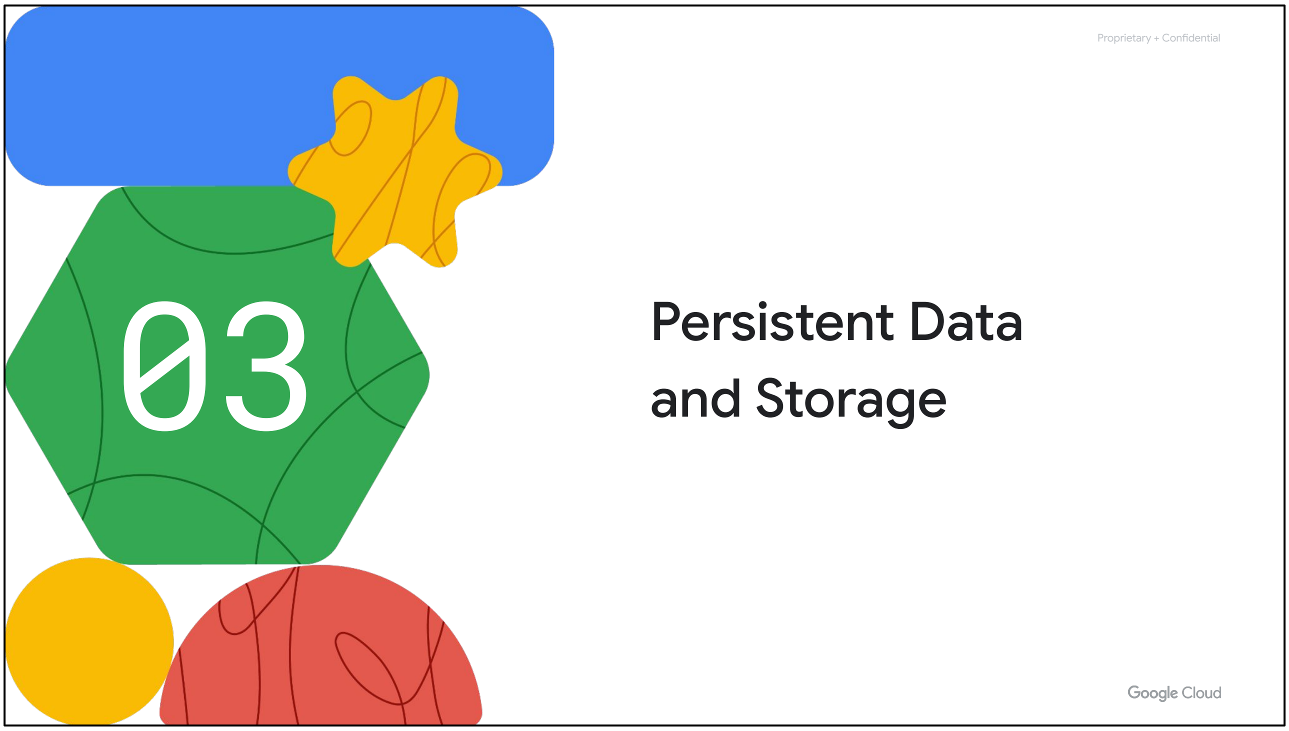


Proprietary + Confidential

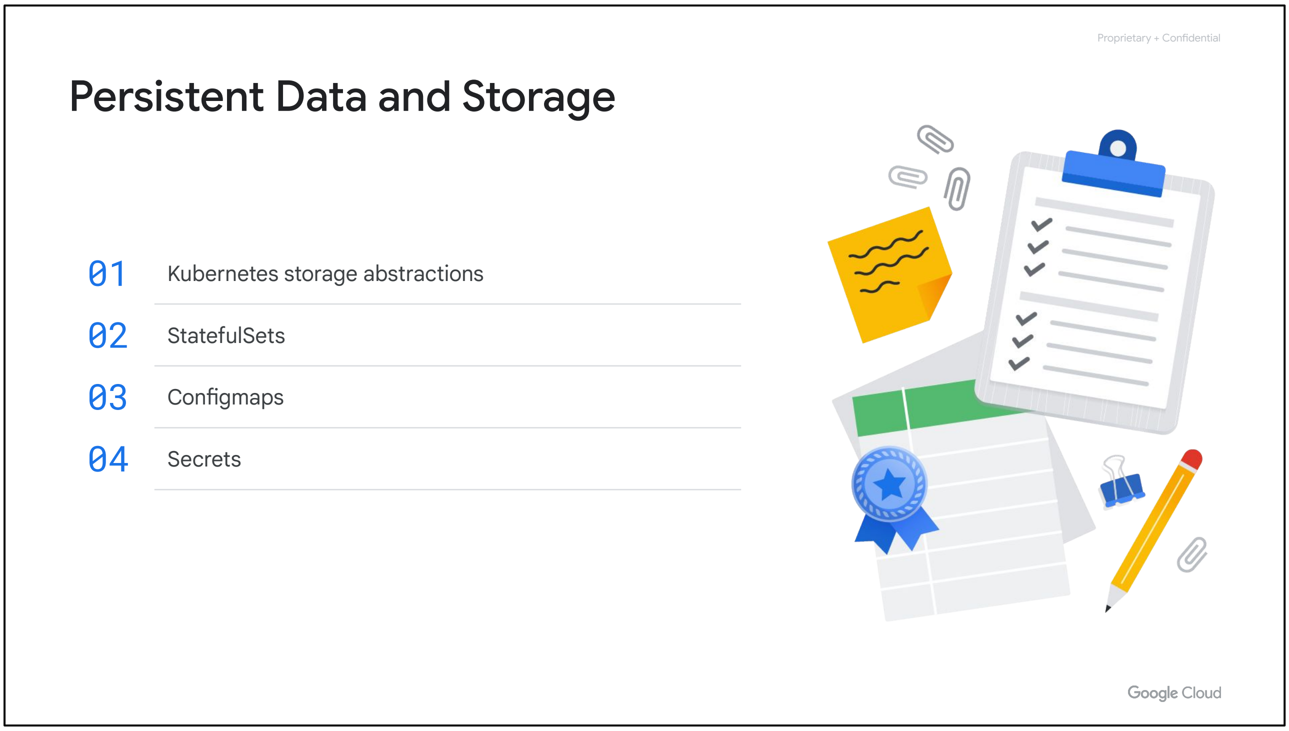
03 Persistent Data and Storage

Google Cloud

As Kubernetes Pods are ephemeral by design, the way GKE handles data storage is a little different than the way Compute Engine instances do. As a result, it's important to ensure the application's data remains present even if a Pod is updated or replaced.

For this to happen, StatefulSets can be used. StatefulSets are a Kubernetes controller that manages the lifecycle of Pods that require persistent storage.

ConfigMaps are a Kubernetes API object that stores configuration data. They can be used during application deployment to decouple configuration artifacts from container definitions. To keep sensitive information from accidental exposure Kubernetes Secrets can be used.

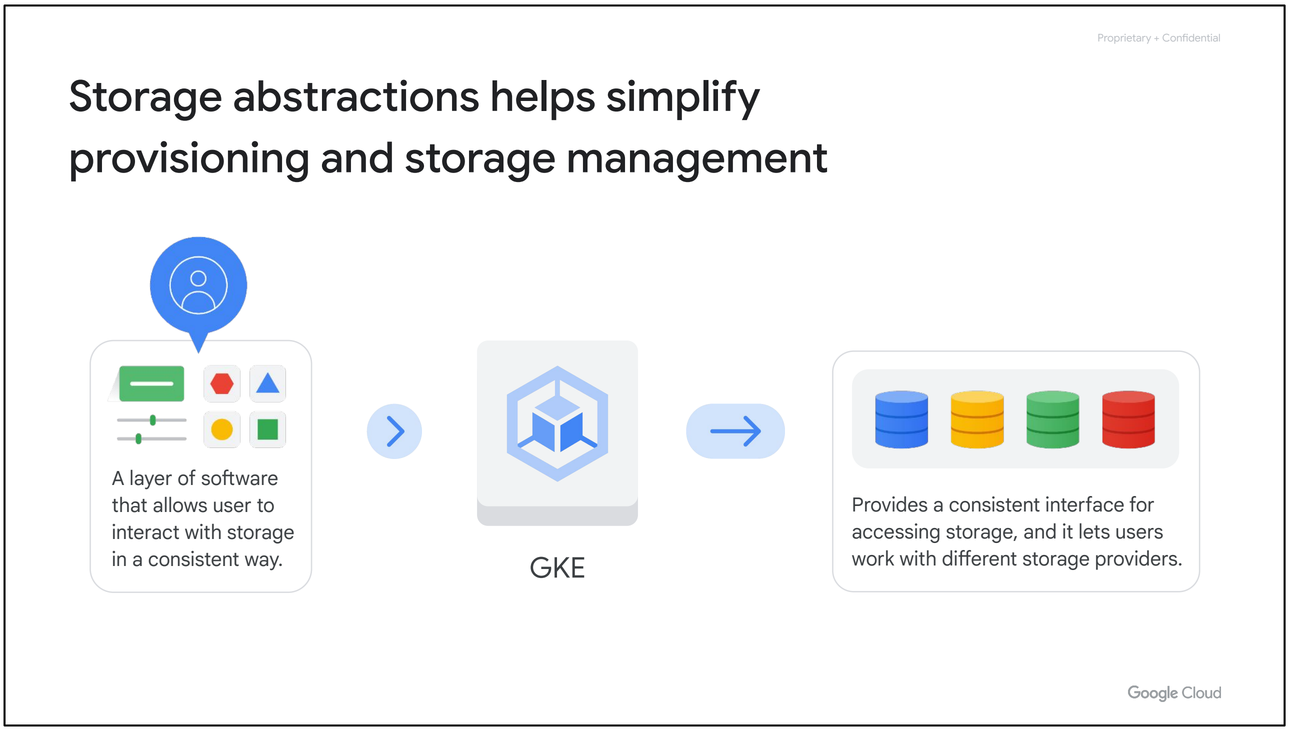


Persistent Data and Storage

In this section of the course titled, Persistent Data and Storage, you'll:

- Explore and practice working with Kubernetes storage abstractions.
- Identify ephemeral and durable Volumes.
- See how to run and maintain sets of Pods by using StatefulSets.
- Examine how ConfigMaps decouple configuration from Pods.
- And manage and store sensitive access and authentication data with Secrets.

Persistent Data and Storage

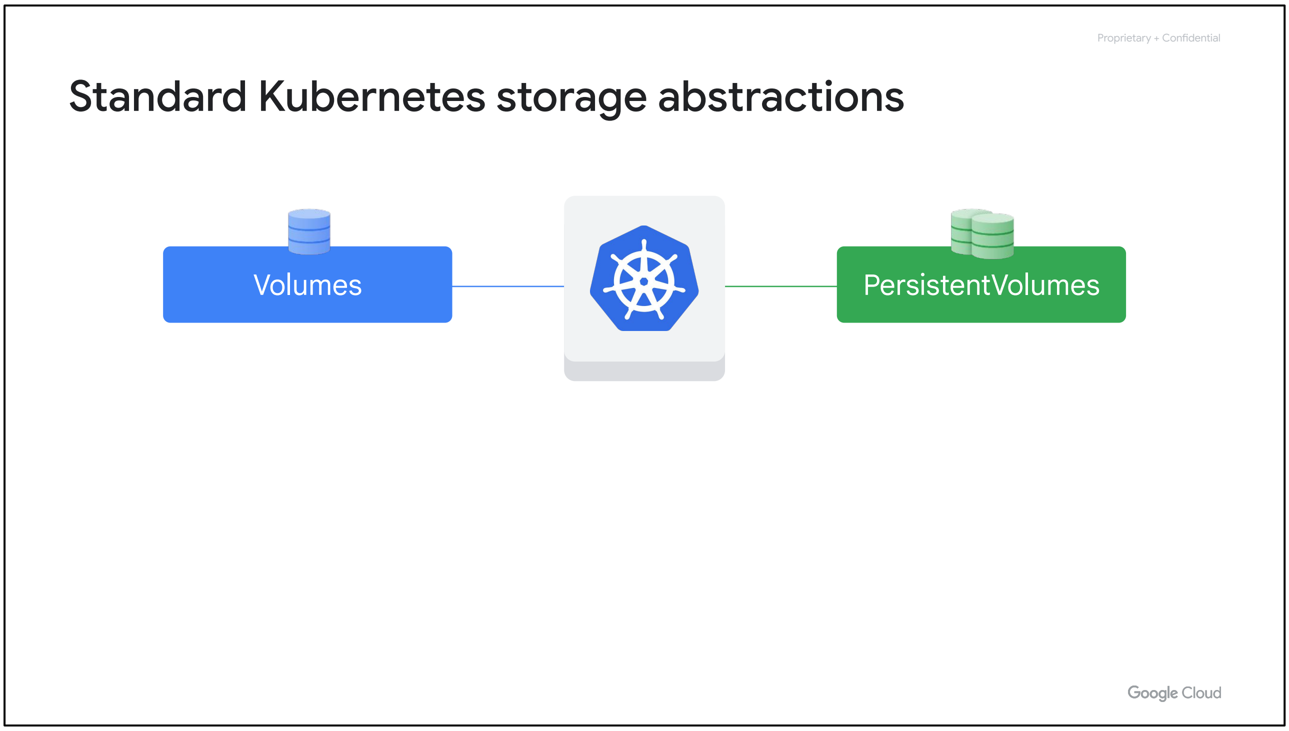


Storage abstractions helps simplify provisioning and storage management

A layer of software that allows user to interact with storage in a consistent way.
GKE
Provides a consistent interface for accessing storage, and it lets users work with different storage providers.

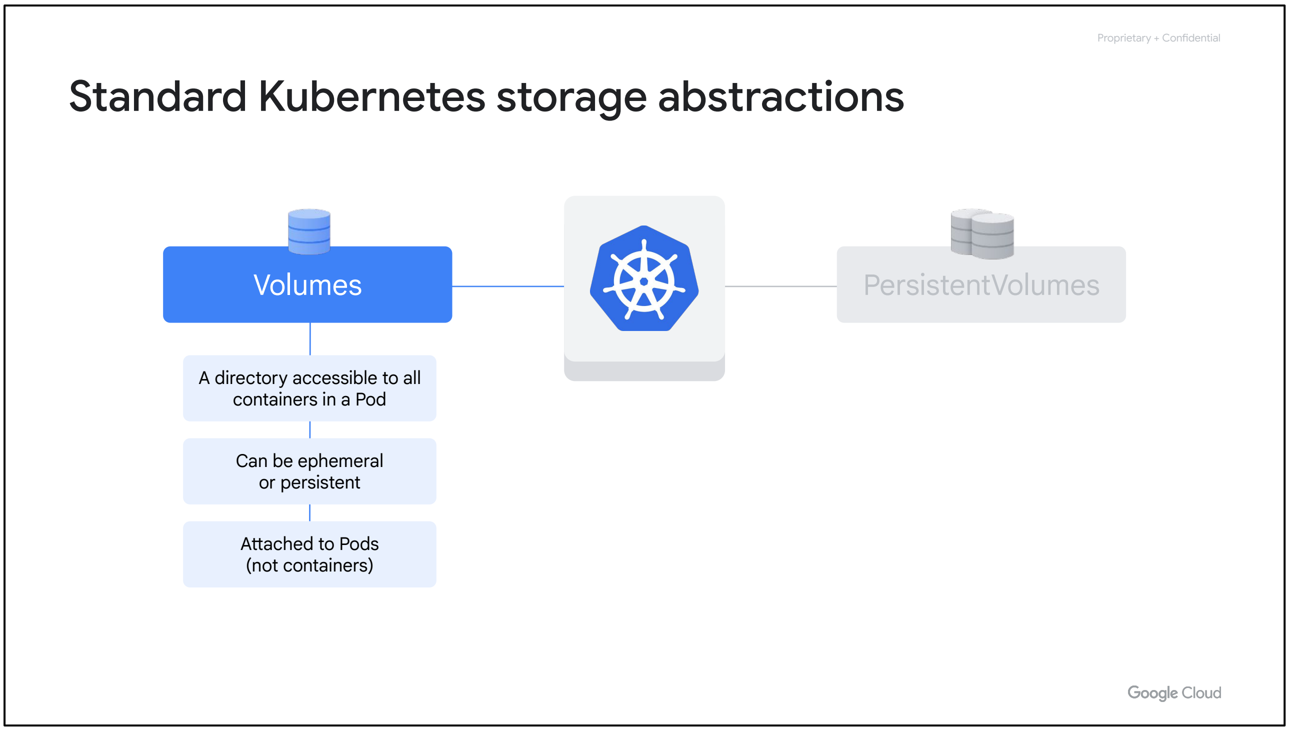
In GKE, a storage abstraction is a layer of software that allows you to interact with storage in a consistent way, regardless of who the underlying storage provider is.

This abstraction layer is important because it helps simplify the process of provisioning and managing storage, it provides a consistent interface for accessing storage, and it lets you use different storage providers so you have more flexibility in choosing the right storage for your needs.



Standard Kubernetes storage abstractions

The standard storage abstractions that Kubernetes provides are Volumes and PersistentVolumes. Let’s examine the difference between the two and how they can be used to store and share information between Pods.



Proprietary + Confidential

Standard Kubernetes storage abstractions

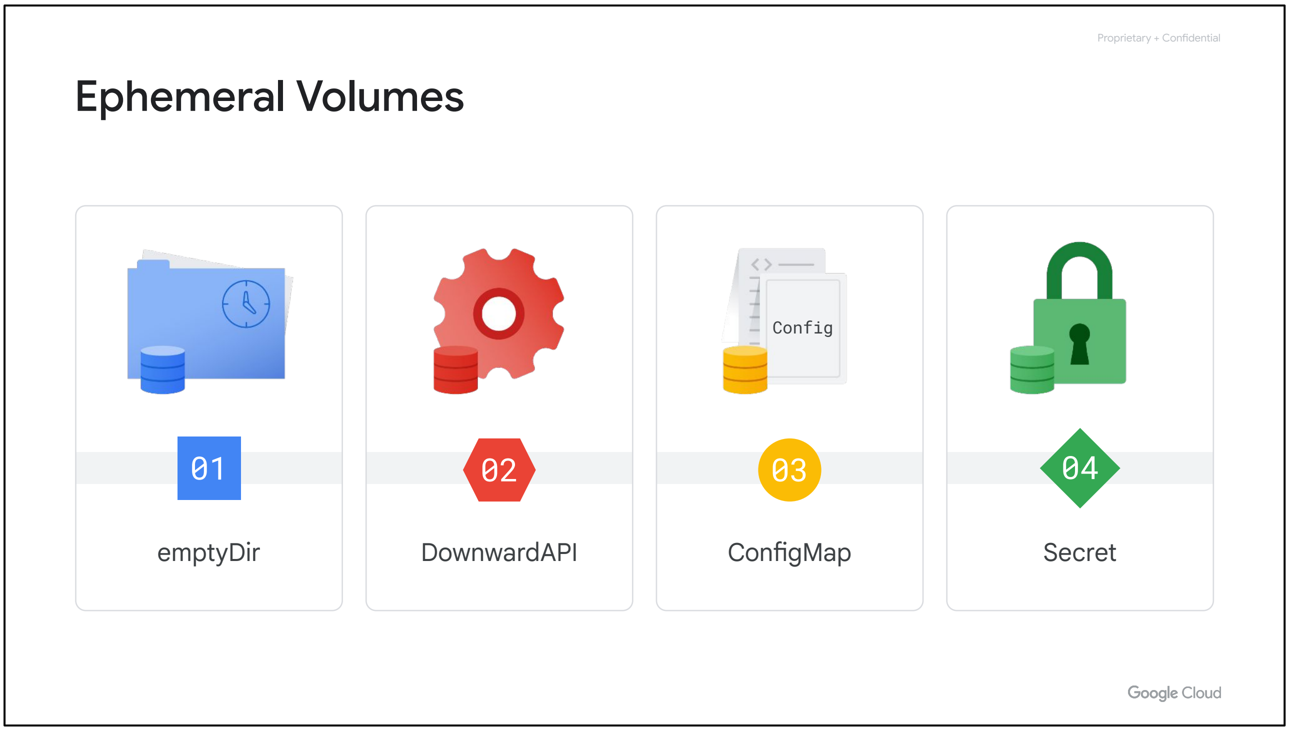
Volumes
A directory accessible to all containers in a Pod
Can be ephemeral or persistent
Attached to Pods (not containers)
PersistentVolumes

Google Cloud

We'll begin with Volumes. Kubernetes uses objects to represent the resources it manages, and this applies to both storage and Pods.

- A Volume is a directory accessible to all containers in a Pod.
- Some Volumes are ephemeral, which means they last only as long as the Pod to which they are attached.
- However, some Volumes are persistent, which means that they can outlive a Pod.

Volumes are attached to Pods, not containers. This is important to understand because if a Pod isn't mapped to a node, then the Volume won't be either.



Proprietary + Confidential

Ephemeral Volumes

01 emptyDir

02 DownwardAPI

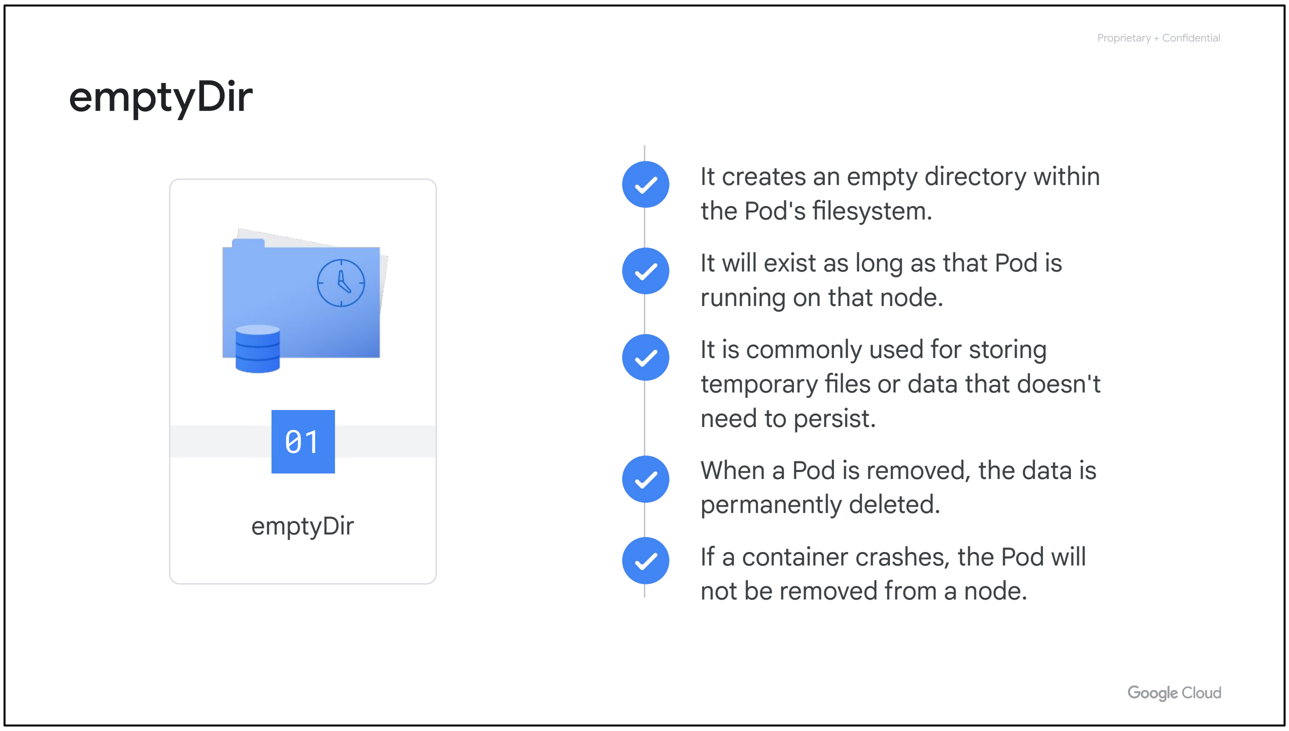
03 ConfigMap

04 Secret

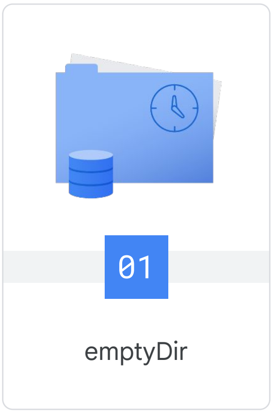
Google Cloud

Kubernetes offers a variety of Volume types, both ephemeral and durable.

Let's start by looking at ephemeral Volumes, including emptyDir, DownwardAPI, ConfigMap, and Secret.



emptyDir

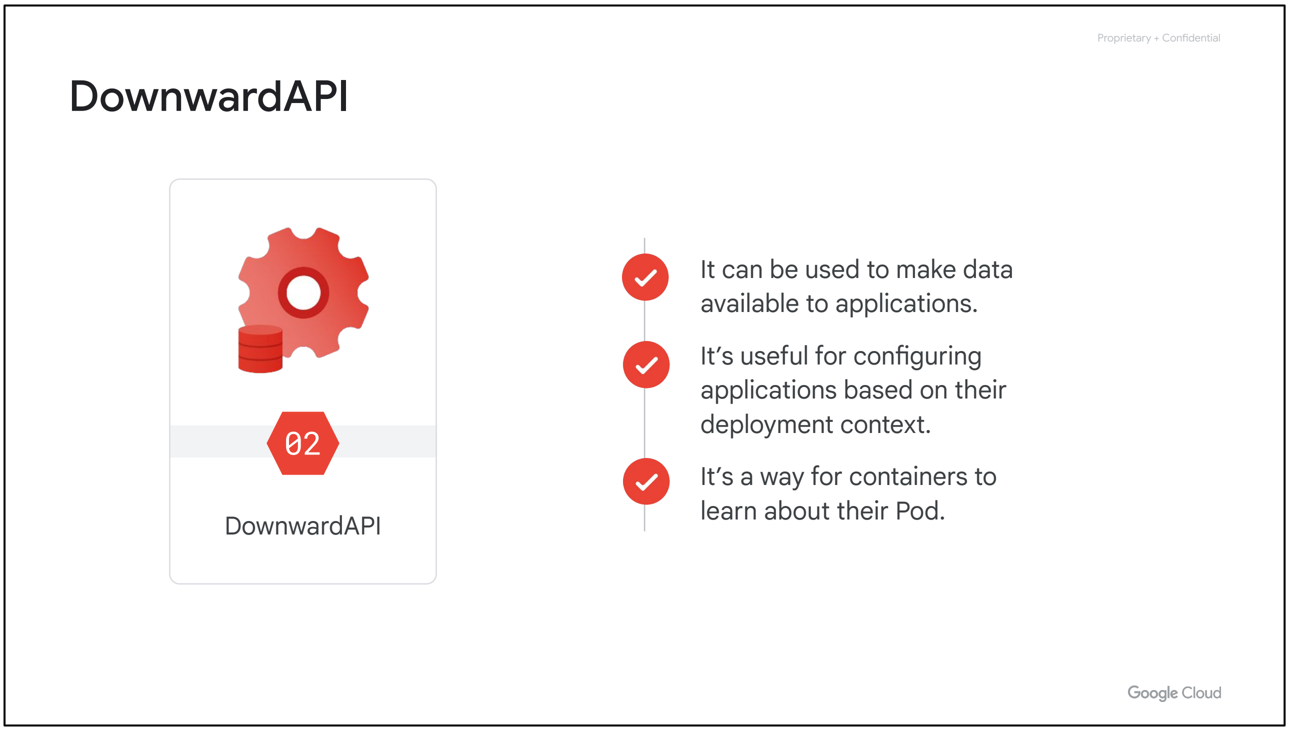


- ✓ It creates an empty directory within the Pod's filesystem.
- ✓ It will exist as long as that Pod is running on that node.
- ✓ It is commonly used for storing temporary files or data that doesn't need to persist.
- ✓ When a Pod is removed, the data is permanently deleted.
- ✓ If a container crashes, the Pod will not be removed from a node.

emptyDir is the most basic type of ephemeral Volume. It creates an empty directory within the Pod's filesystem to read and write from, and will exist as long as that Pod is running on that node.

emptyDir is commonly used for storing temporary files or data that doesn't need to persist beyond the Pod's lifetime. Examples might include 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.

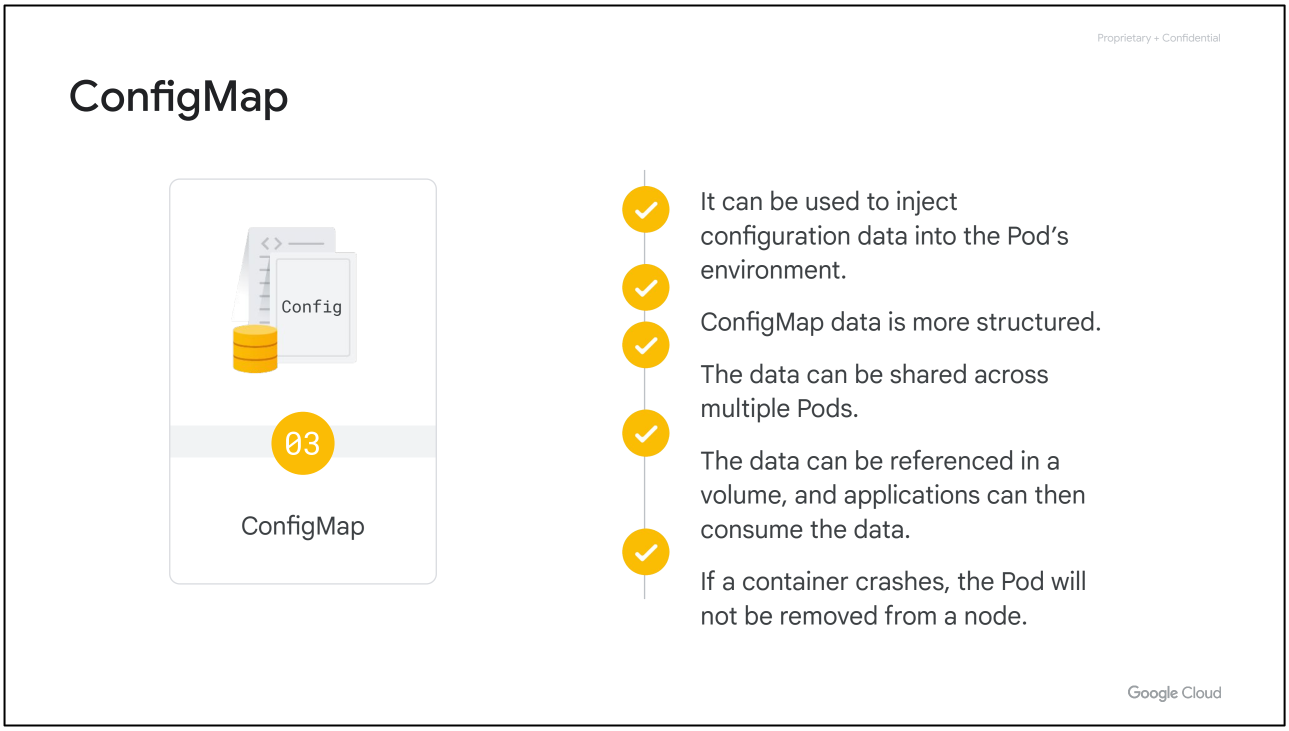
When a Pod is removed from a node for any reason, the data in the emptyDir is permanently deleted. However, if a container crashes, that event will not cause a Pod to be removed from a node. This means the data in the emptyDir volume will remain safe.



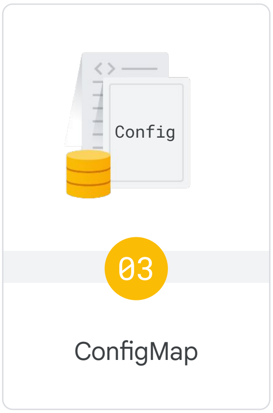
DownwardAPI

- ✓ It can be used to make data available to applications.
- ✓ It's useful for configuring applications based on their deployment context.
- ✓ It's a way for containers to learn about their Pod.

Next is the **DownwardAPI** Volume type, which can be used to make data from the Downward API available to applications. This data can include Pod labels, annotations, secrets, and node information, which makes it useful for configuring applications based on their deployment context. It's a way for containers to learn about their Pod.



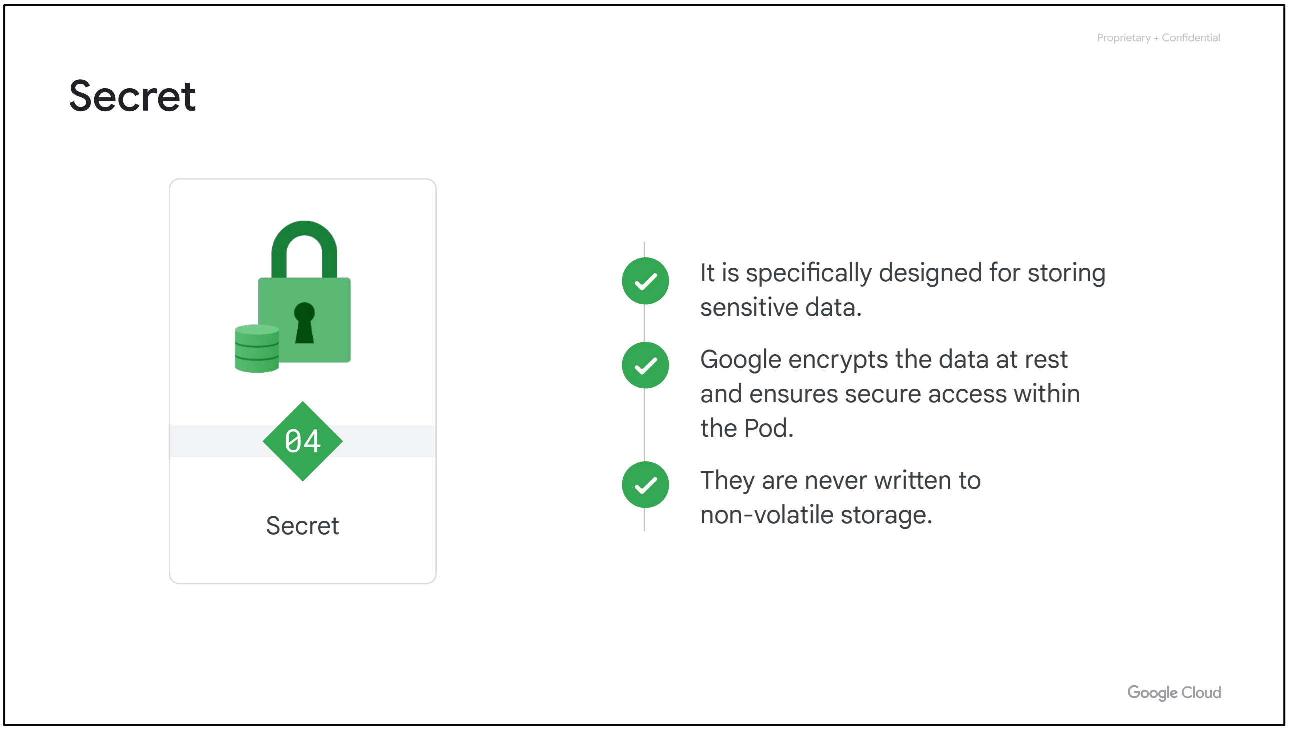
ConfigMap



- ✓ It can be used to inject configuration data into the Pod's environment.
- ✓ ConfigMap data is more structured.
- ✓ The data can be shared across multiple Pods.
- ✓ The data can be referenced in a volume, and applications can then consume the data.
- ✓ If a container crashes, the Pod will not be removed from a node.

Then there are **ConfigMap** Volumes. Similar to DownwardAPI, ConfigMap Volumes can be used to inject configuration data into the Pod's environment. However, ConfigMap data is more structured, is stored as key-value pairs, and can be shared across multiple Pods.

The data stored in a ConfigMap object can be referenced in a volume, as if it were a tree of files and directories. Applications can then consume the data.

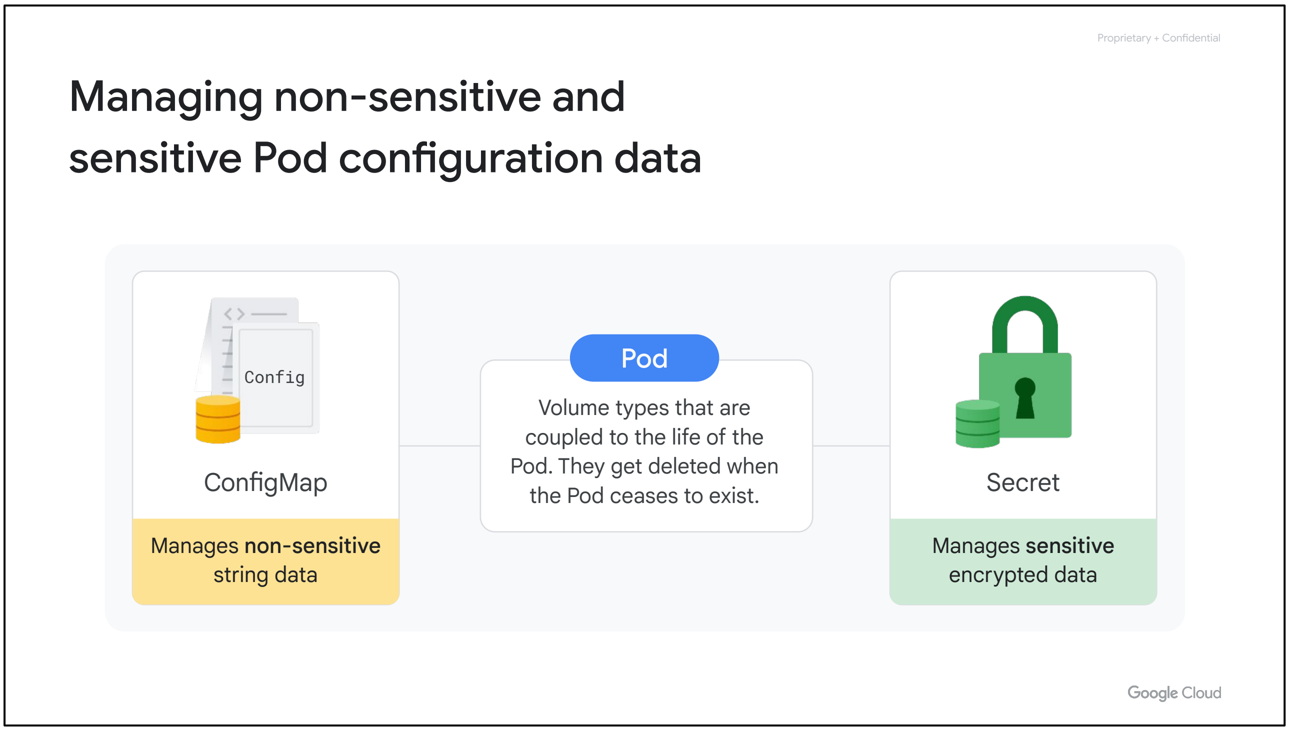


Secret

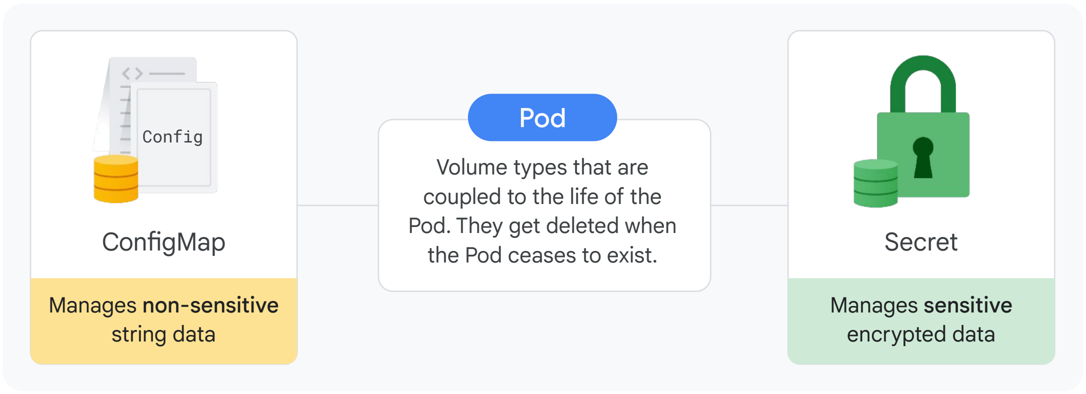
- ✓ It is specifically designed for storing sensitive data.
- ✓ Google encrypts the data at rest and ensures secure access within the Pod.
- ✓ They are never written to non-volatile storage.

Finally, there is the **Secret** Volume type, which is specifically designed for storing sensitive data, such as passwords, tokens, or API keys. Secrets are unencrypted, but Google encrypts the data at rest and ensures secure access within the Pod.

Secret Volumes are backed by in-memory file systems, so the Secrets are never written to non-volatile storage.



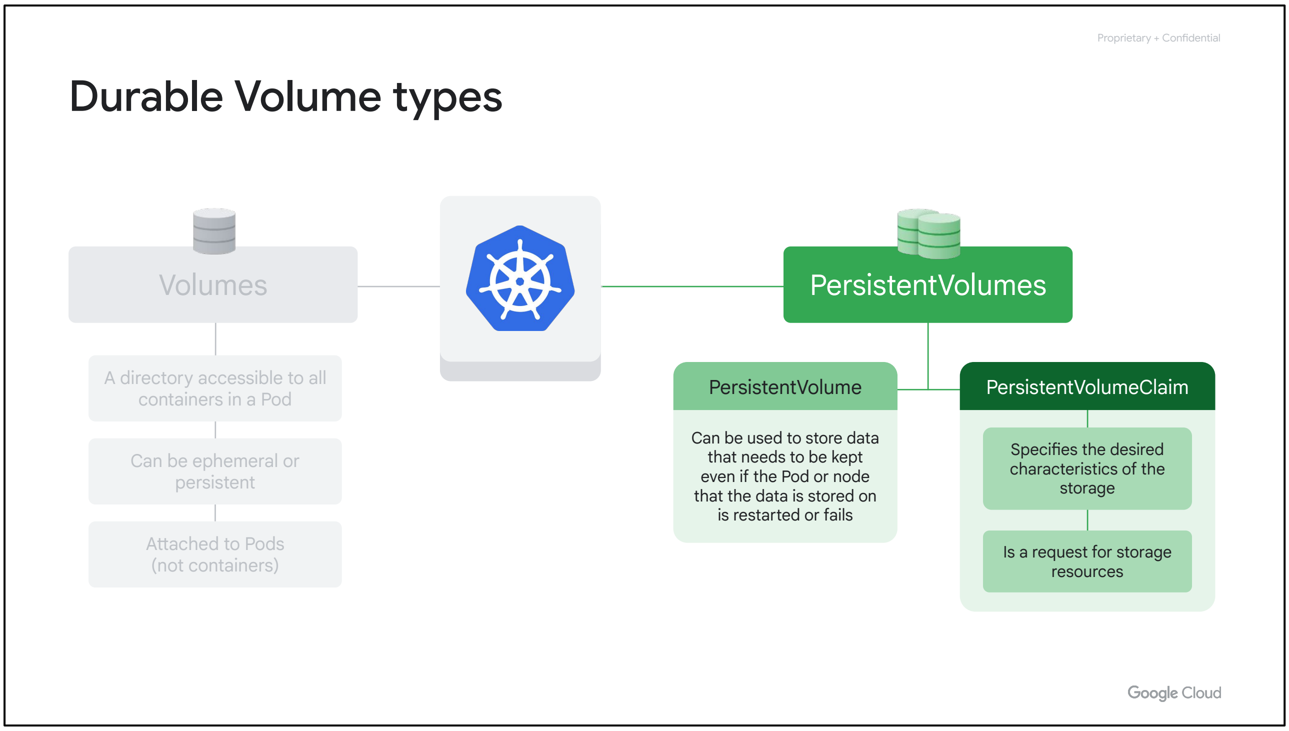
Managing non-sensitive and sensitive Pod configuration data


Config
ConfigMap
Manages non-sensitive string data
Pod
Volume types that are coupled to the life of the Pod. They get deleted when the Pod ceases to exist.
Secret
Manages sensitive encrypted data

Secrets aren't secret just because of the way they are configured. Differentiating between ConfigMaps and Secrets provides a way to manage non-sensitive and sensitive Pod configuration data differently.

Some Volume types, like Secrets and ConfigMaps, are coupled to the life of the Pod and and are deleted when the Pod ceases to exist. But it's important to note that although the Secret and ConfigMap Volumes that attach to individual Pods are ephemeral, the objects are not.

At a fundamental level, ConfigMap, Secret, and DownwardAPI allow for different kinds of Kubernetes data into a Pod.


Proprietary + Confidential

Durable Volume types

Volumes
A directory accessible to all containers in a Pod
Can be ephemeral or persistent
Attached to Pods (not containers)
PersistentVolumes
PersistentVolume
Can be used to store data that needs to be kept even if the Pod or node that the data is stored on is restarted or fails
PersistentVolumeClaim
Specifies the desired characteristics of the storage
Is a request for storage resources
Google Cloud

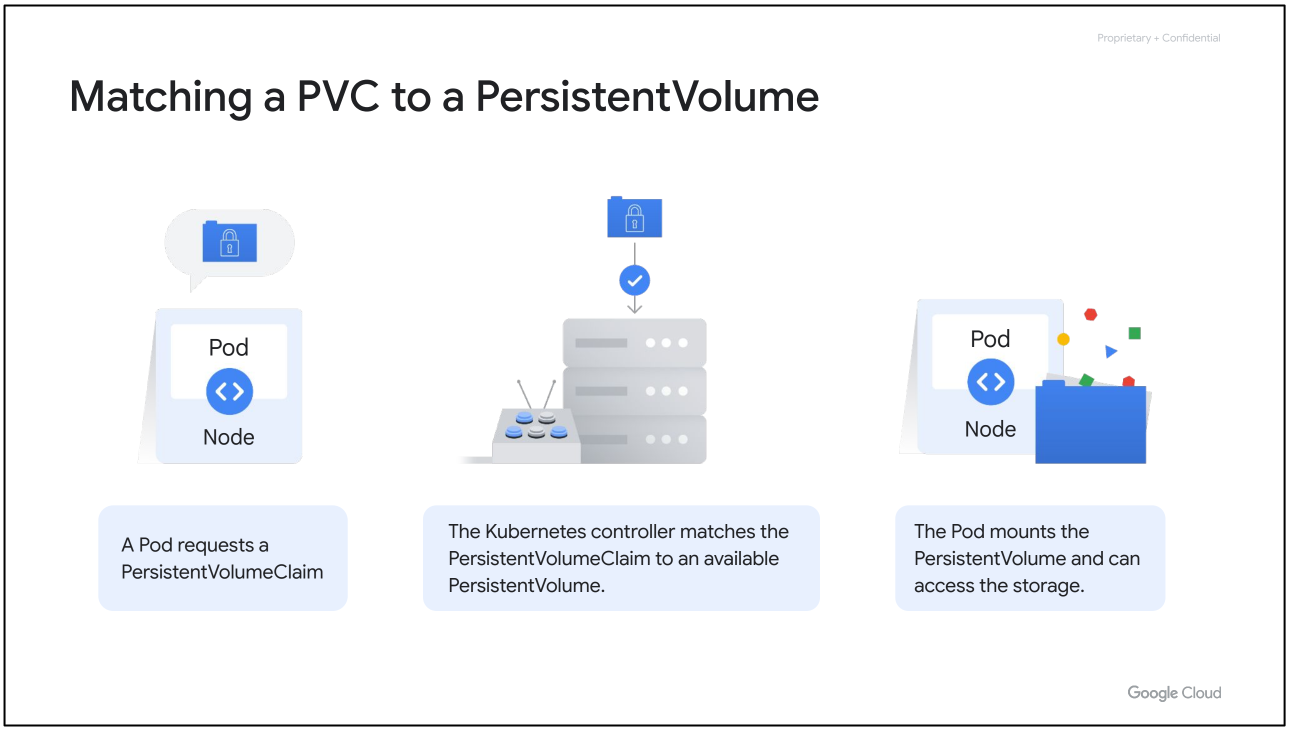
Along with ephemeral Volume types, Kubernetes offers a durable Volume type: PersistentVolume. The PersistentVolume abstraction has two components: PersistentVolume, or PV, and PersistentVolumeClaim, or PVC. Let's define both, starting with PersistentVolume.

PersistentVolume is:

- A type of storage that can be used to store data that needs to be kept even if the Pod or node that the data is stored on is restarted or fails. This is in contrast to ephemeral storage, which is only available while the Pod or node is running.
- PersistentVolumes are typically used to store data that is important to the application, such as databases, file systems, and configuration files.

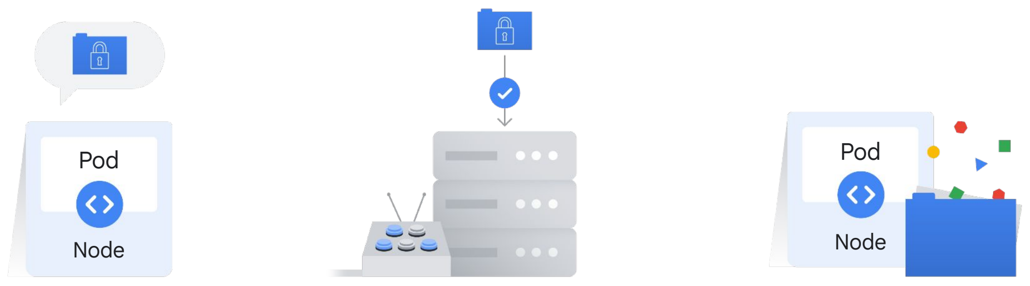
That brings us to a **PersistentVolumeClaim**, or PVC, which:

- Represents a request for storage resources.
- It's a way for users to request storage that can be used by Pods to store data that needs to persist across Pod restarts or node failures. A PVC specifies the desired characteristics of the storage, such as the size, access mode, and storage class.



Proprietary + Confidential

Matching a PVC to a PersistentVolume



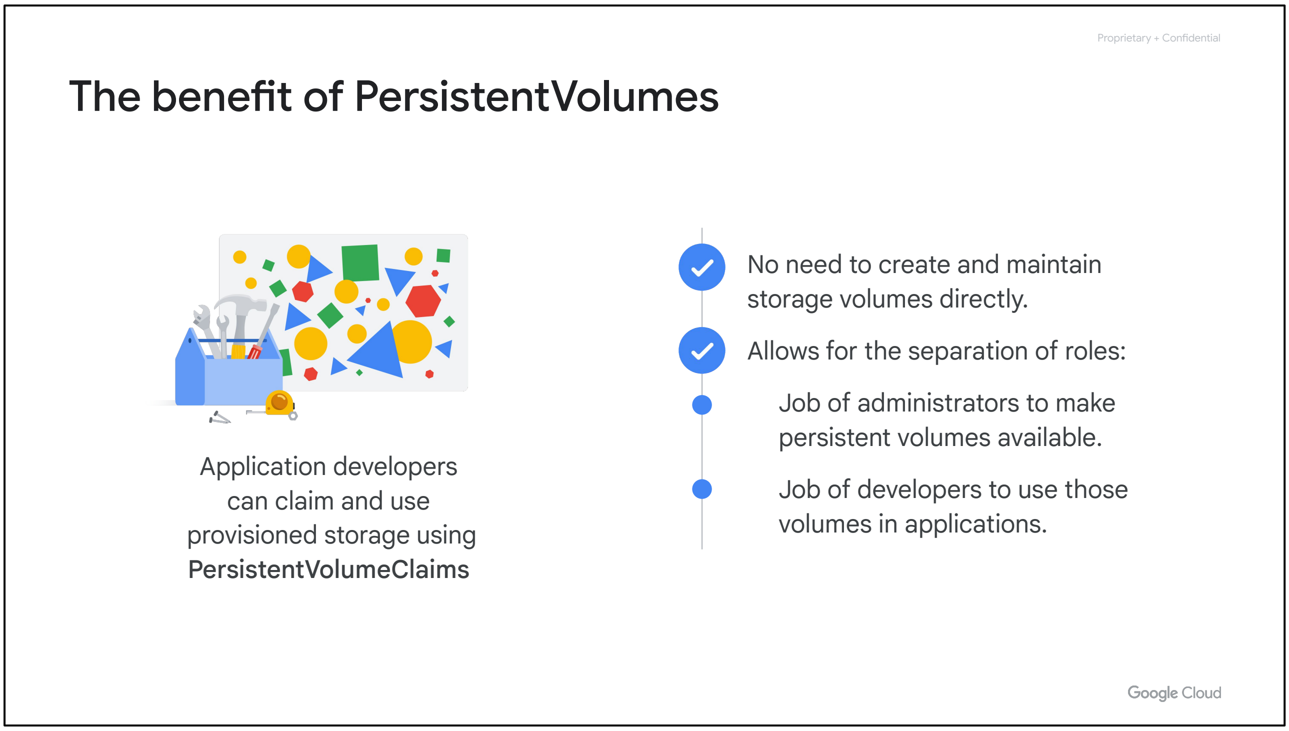
A Pod requests a PersistentVolumeClaim

The Kubernetes controller matches the PersistentVolumeClaim to an available PersistentVolume.

The Pod mounts the PersistentVolume and can access the storage.

Google Cloud

When a Pod requests a PVC, the Kubernetes controller matches the PVC to an available PersistentVolume that meets the requirements of the PVC. Once a PersistentVolume is matched to a PVC, the Pod can mount the PersistentVolume and access the storage.



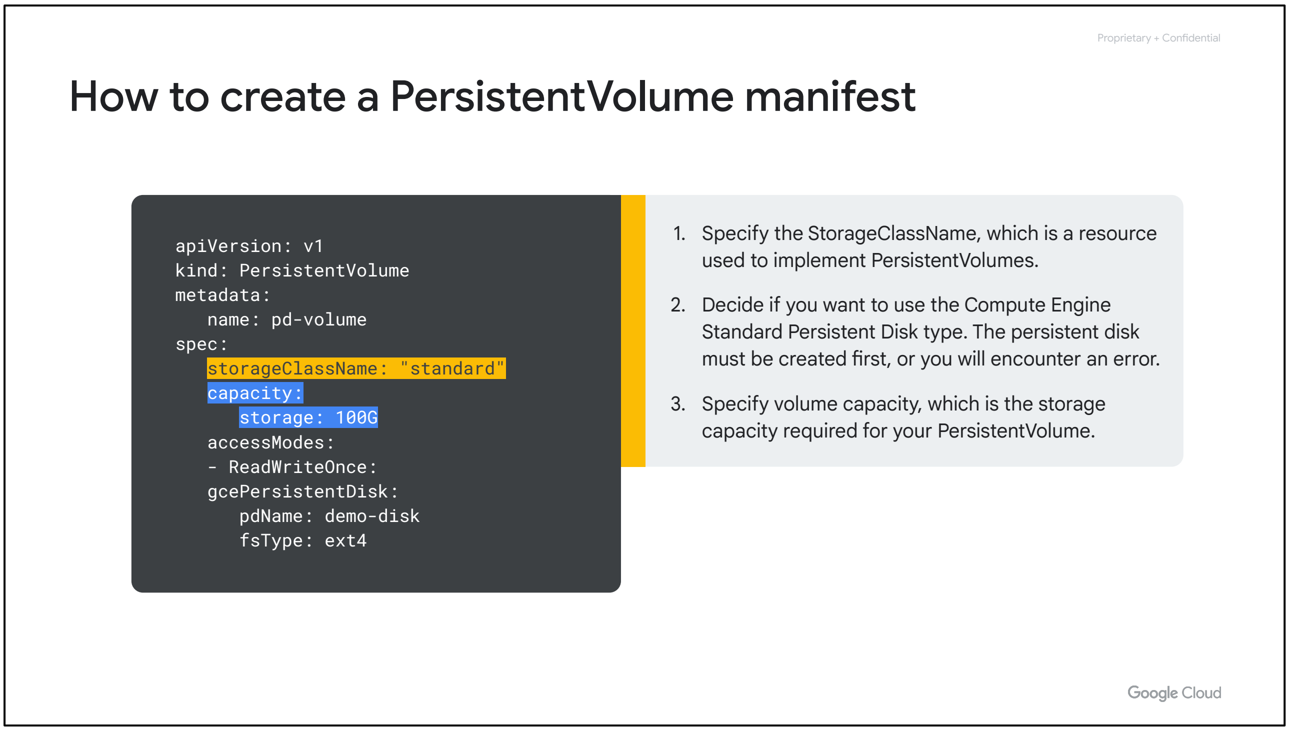
The benefit of PersistentVolumes

Application developers can claim and use provisioned storage using **PersistentVolumeClaims**

- ✓ No need to create and maintain storage volumes directly.
- ✓ Allows for the separation of roles:
  - Job of administrators to make persistent volumes available.
  - Job of developers to use those volumes in applications.

Application developers can claim and use provisioned storage using PersistentVolumeClaims without creating and maintaining storage volumes directly.

This allows for the separation of roles, as it's the job of administrators to make persistent volumes available, and the job of developers to use those volumes in applications.



How to create 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
```

1. Specify the StorageClassName, which is a resource used to implement PersistentVolumes.
2. Decide if you want to use the Compute Engine Standard Persistent Disk type. The persistent disk must be created first, or you will encounter an error.
3. Specify volume capacity, which is the storage capacity required for your PersistentVolume.

Google Cloud

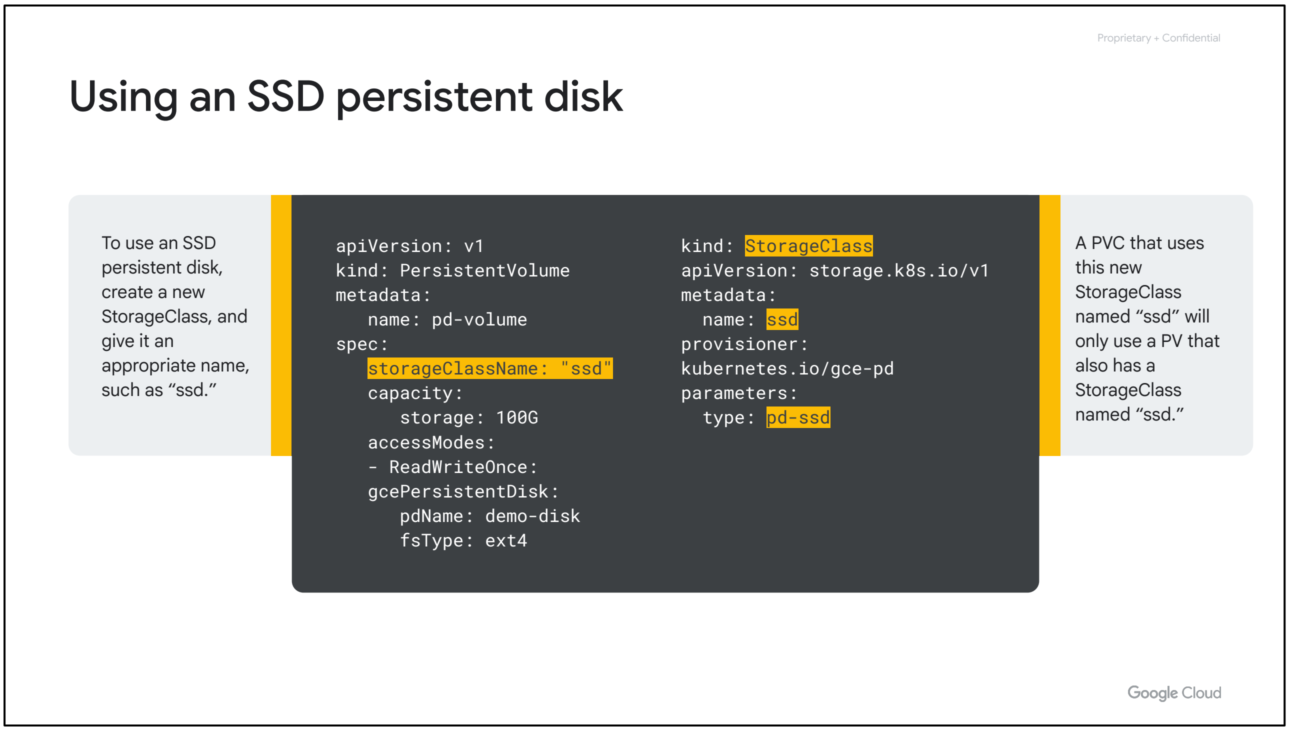
Now let's shift focus and explore how to create a PersistentVolume manifest.

The first task is to specify the **StorageClassName**, which is a resource used to implement PersistentVolumes.

When you define a PVC in a Pod, the PVC uses the StorageClassName. This means that the PV StorageClassName must match the PVC StorageClassName if you want the claim to be successful.

The next step is to decide if you want to use the Compute Engine Standard Persistent Disk type. If so, GKE has a default StorageClass name **'standard'** that can be used. The persistent disk must be created first, or you will encounter an error. With GKE clusters, a PVC with no defined StorageClass will use this default StorageClass, and it will provide storage by using a standard Persistent Disk.

From there, specify volume capacity, which is the storage capacity required for your PersistentVolume.



Using an SSD persistent disk

To use an SSD persistent disk, create a new StorageClass, and give it an appropriate name, such as “ssd.”

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

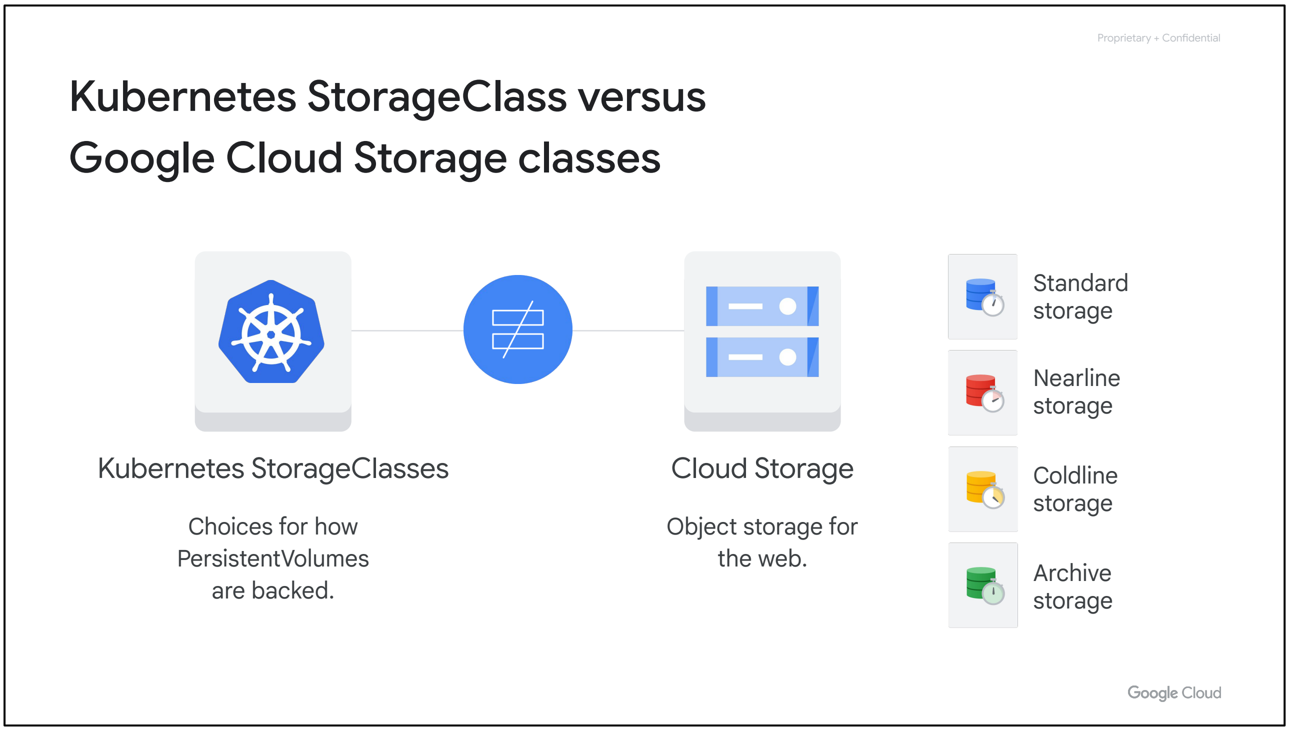
```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: ssd
provisioner:
kubernetes.io/gce-pd
parameters:
  type: pd-ssd
```

A PVC that uses this new StorageClass named “ssd” will only use a PV that also has a StorageClass named “ssd.”

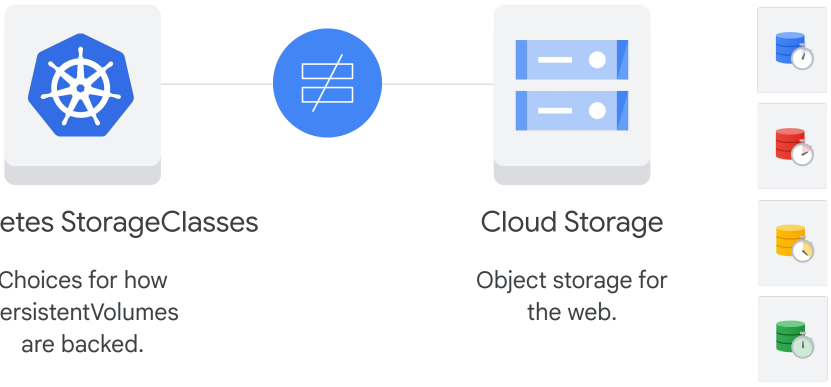
However, if you want to use an SSD persistent disk, create a new StorageClass, and give it an appropriate name, such as “SSD.” In the parameters section, define the type as **pd-ssd**.

Remember, A PVC that uses this new StorageClass named SSD will only use a PV that also has a StorageClass named SSD.

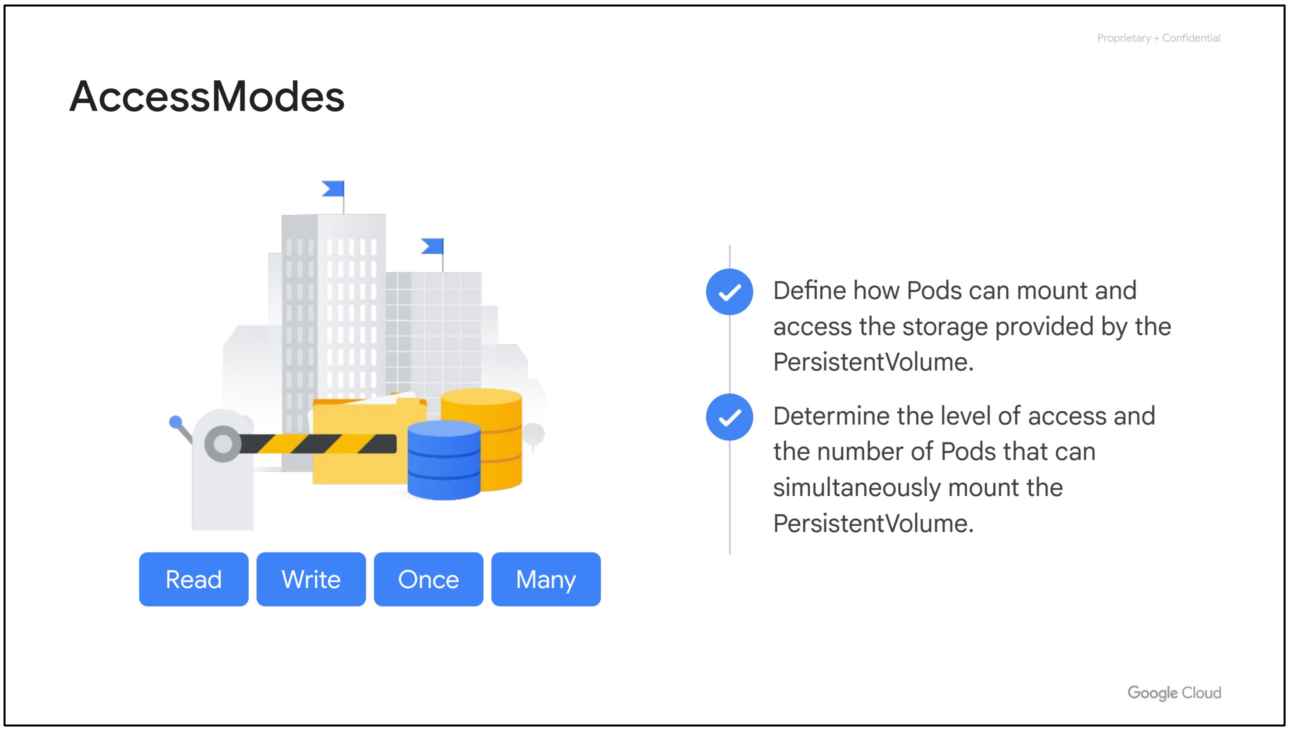
And when a new storage class is created, it can be viewed in the Cloud Console.



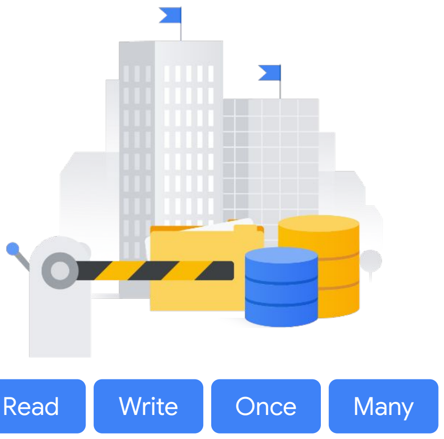
Kubernetes StorageClass versus Google Cloud Storage classes


Kubernetes StorageClasses
Choices for how PersistentVolumes are backed.
Cloud Storage
Object storage for the web.
Standard storage
Nearline storage
Coldline storage
Archive storage

Now it’s important to note that Kubernetes StorageClasses and Google Cloud Storage classes are not the same thing. Google Cloud Storage provides object storage for the web, whereas Kubernetes StorageClasses are choices for how PersistentVolumes are backed.

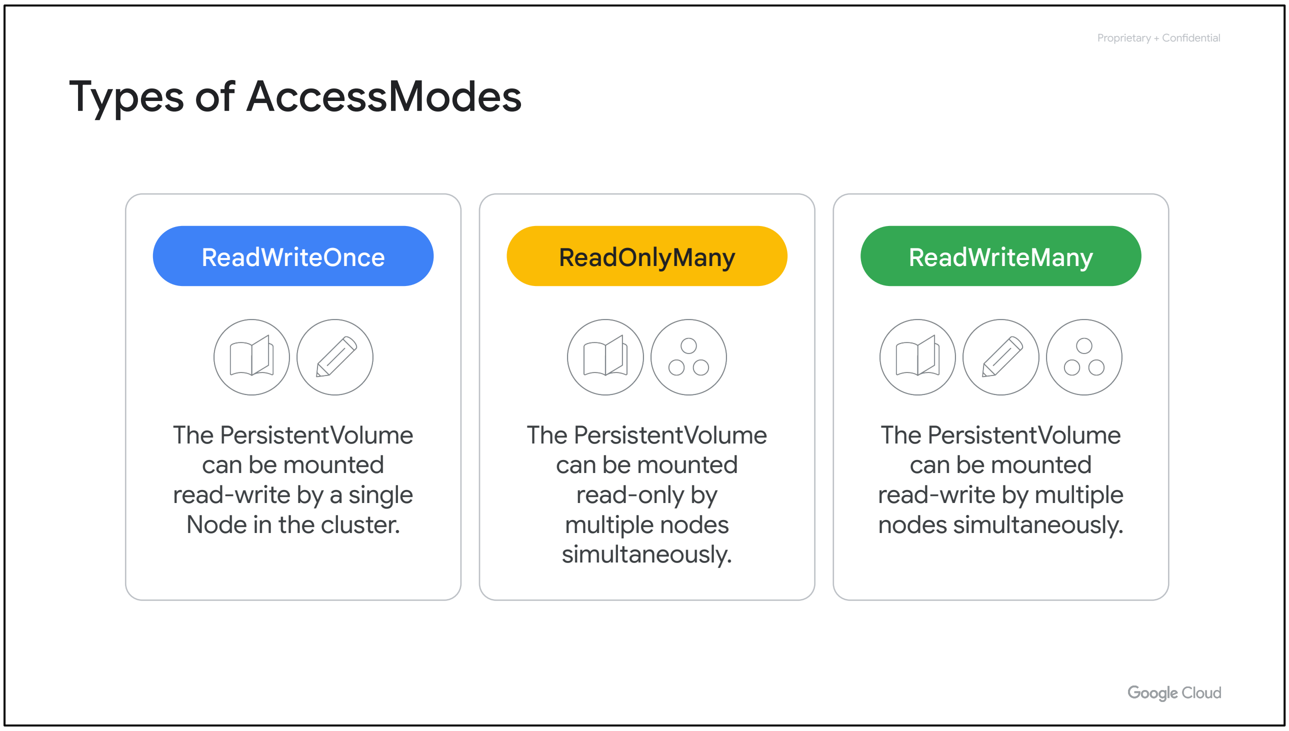


AccessModes



- ✓ Define how Pods can mount and access the storage provided by the PersistentVolume.
- ✓ Determine the level of access and the number of Pods that can simultaneously mount the PersistentVolume.

Then there are **AccessModes**. AccessModes in PersistentVolumes define how Pods can mount and access the storage provided by the PV. They determine the level of access and the number of Pods that can simultaneously mount the PV.



Proprietary + Confidential

Types of AccessModes

ReadWriteOnce	ReadOnlyMany	ReadWriteMany
The PersistentVolume can be mounted read-write by a single Node in the cluster.	The PersistentVolume can be mounted read-only by multiple nodes simultaneously.	The PersistentVolume can be mounted read-write by multiple nodes simultaneously.

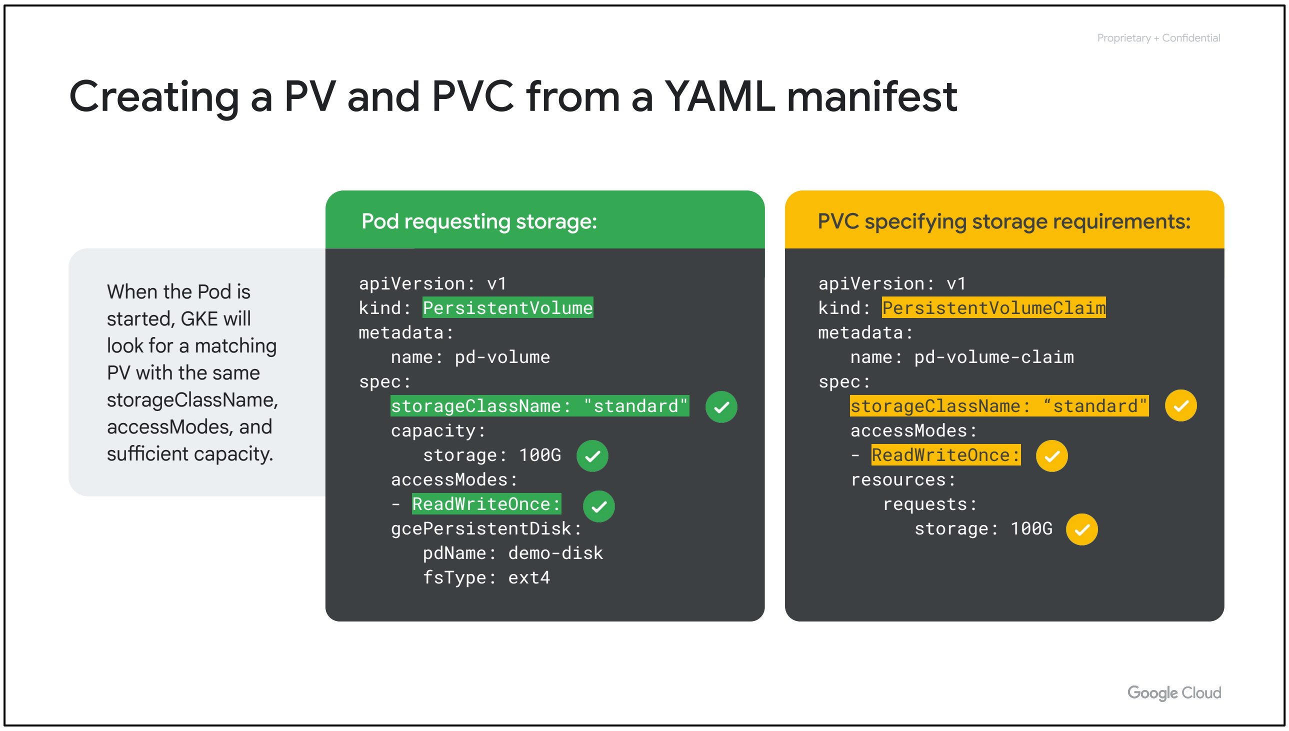
Google Cloud

There are different types of AccessModes.

With **ReadWriteOnce**, the PV can be mounted read-write by a single Node in the cluster. This means that any Pod running on that Node can access the PV for reading and writing. For most applications, persistent disks are mounted as ReadWriteOnce.

With **ReadOnlyMany**, the PV can be mounted read-only by multiple nodes simultaneously. This means that Pods running on these nodes can only read data from the PV, not write to it.

And with **ReadWriteMany**, the PV can be mounted read-write by multiple nodes simultaneously. Pods running on these nodes can both read and write data to the PV. However, Google Cloud persistent disks do not support ReadWriteMany.



Proprietary + Confidential

Creating a PV and PVC from a YAML manifest

When the Pod is started, GKE will look for a matching PV with the same storageClassName, accessModes, and sufficient capacity.

Pod requesting storage:

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

PVC specifying storage requirements:

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

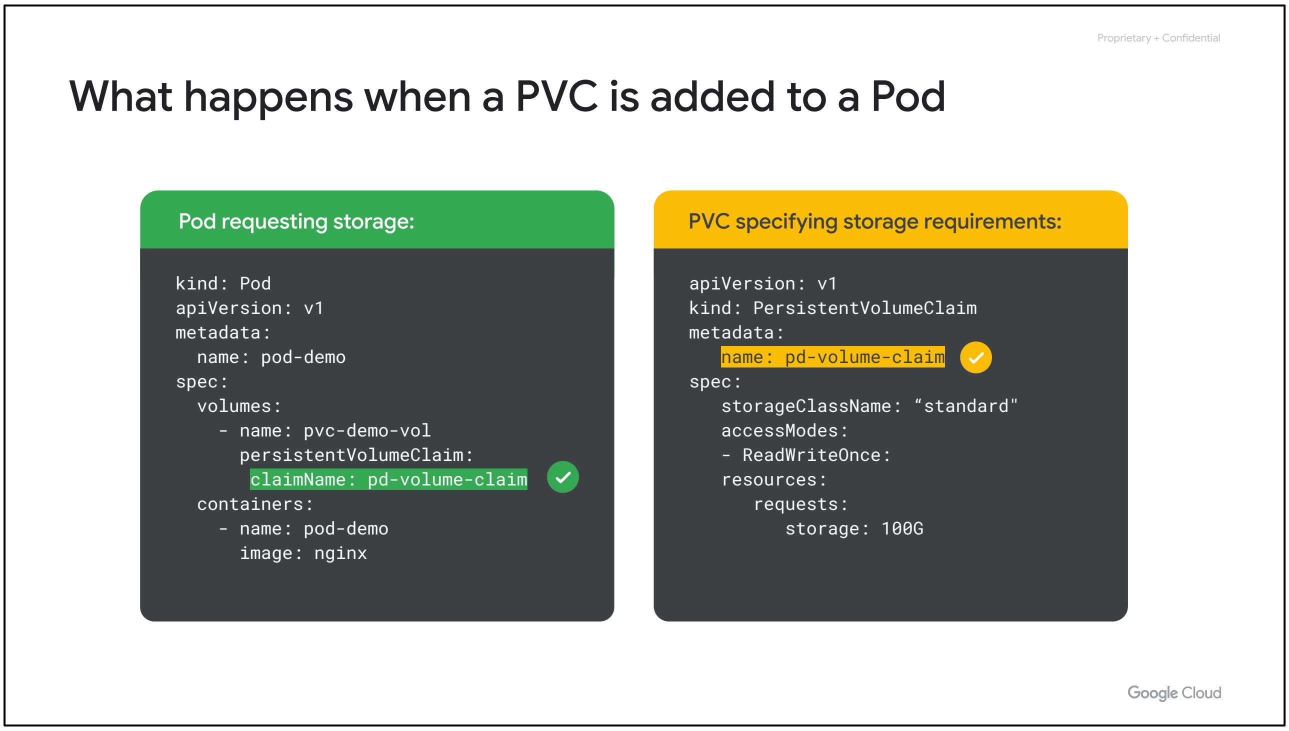
Google Cloud

Persistent Volumes can also be created from a YAML manifest by using a kubectl apply command.

For example, let’s say there is a PV named pd-volume that will have 100 GB of storage allocated and will allow ReadWriteOnce access. It will use a standard Persistent Disk based on the storageClassName, and it will be maintained by cluster administrators.

Remember, PersistentVolumes can’t be added to Pod specifications directly. Instead, PersistentVolumeClaims must be used. 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.



What happens when a PVC is added to a Pod

Pod requesting storage:

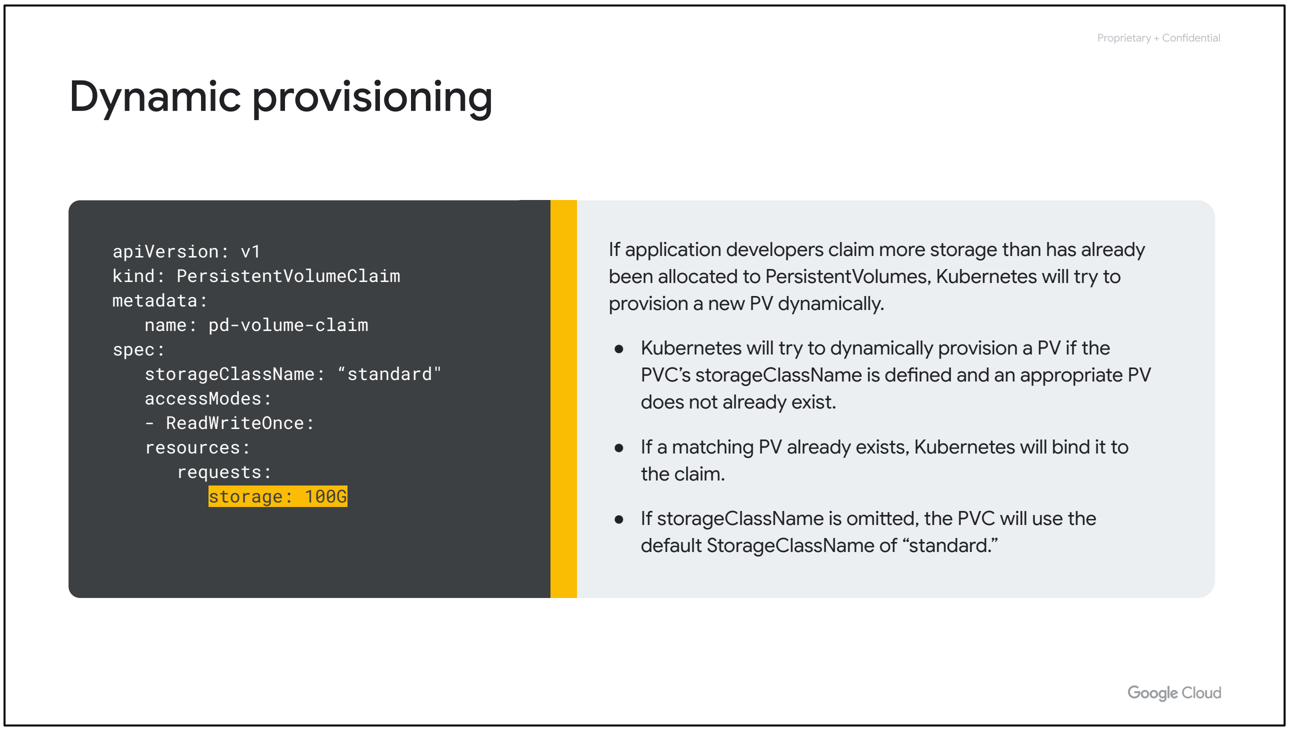
```
kind: Pod
apiVersion: v1
metadata:
  name: pod-demo
spec:
  volumes:
    - name: pvc-demo-vol
      persistentVolumeClaim:
       claimName: pd-volume-claim   ✓
  containers:
    - name: pod-demo
      image: nginx
```

PVC specifying storage requirements:

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

Now let’s explore what happens when a PersistentVolumeClaim is added to a Pod.

When this Pod is started, GKE will look for a matching PV with the same storageClassName, accessModes, and sufficient capacity. The specific cloud implementation doesn’t really matter, because the specific storage that is used to deliver this storage class is controlled by the cluster administrators, not the application developers.



Proprietary + Confidential

Dynamic provisioning

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

If application developers claim more storage than has already been allocated to PersistentVolumes, Kubernetes will try to provision a new PV dynamically.

- Kubernetes will try to dynamically provision a PV if the PVC’s storageClassName is defined and an appropriate PV does not already exist.
- If a matching PV already exists, Kubernetes will bind it to the claim.
- If storageClassName is omitted, the PVC will use the default StorageClassName of “standard.”

Google Cloud

But what if application developers claim more storage than has already been allocated to PersistentVolumes?

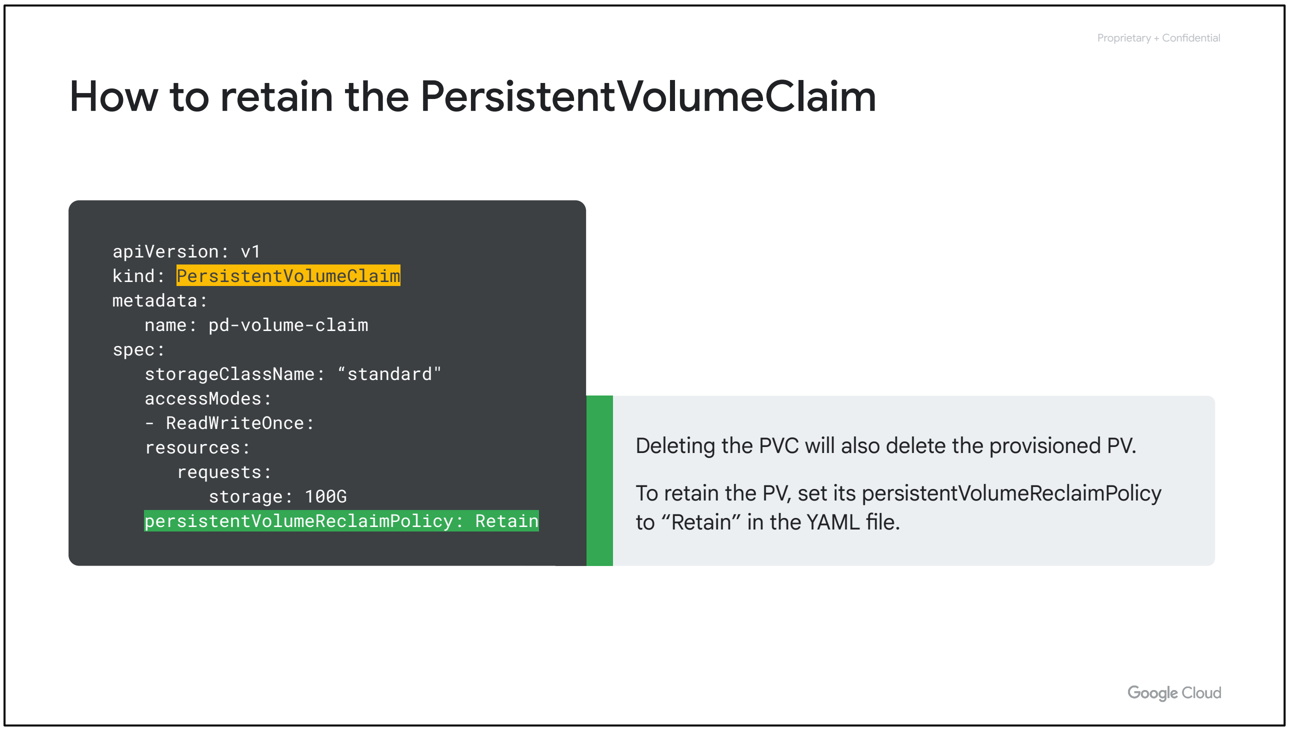
If there isn’t an existing PersistentVolume to satisfy the PersistentVolumeClaim, Kubernetes will try to provision a new one dynamically.

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, Kubernetes will bind it to the claim.

If storageClassName is omitted, the PersistentVolumeClaim will use the default StorageClass, which, in GKE, is named “standard.”

Dynamic provisioning will only work in this case if it is enabled on the cluster.

GKE manages all of this. The application owner does not have to provision the underlying storage, and does not need to embed the details of the underlying storage into the Pod manifest.



How to retain the PersistentVolumeClaim

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

Deleting the PVC will also delete the provisioned PV.

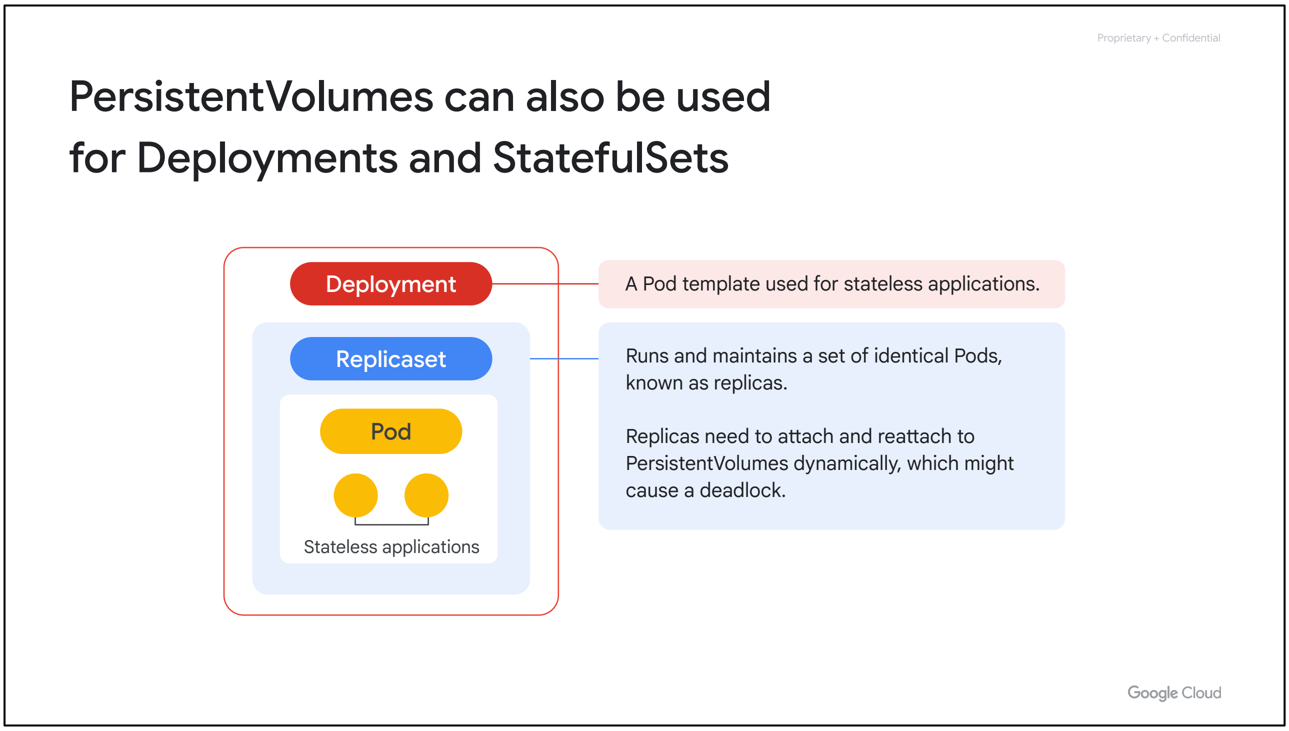
To retain the PV, set its persistentVolumeReclaimPolicy to “Retain” in the YAML file.

Deleting the PersistentVolumeClaim will also delete the provisioned PersistentVolume.

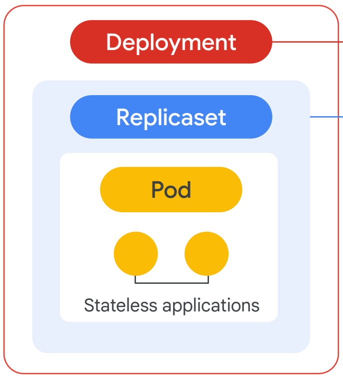
So, if you want to retain the PersistentVolume, set its persistentVolumeReclaimPolicy to Retain in the YAML file.

PersistentVolumeClaims should be deleted when their underlying PersistentVolume is no longer required.

Persistent Data and Storage



PersistentVolumes can also be used for Deployments and StatefulSets



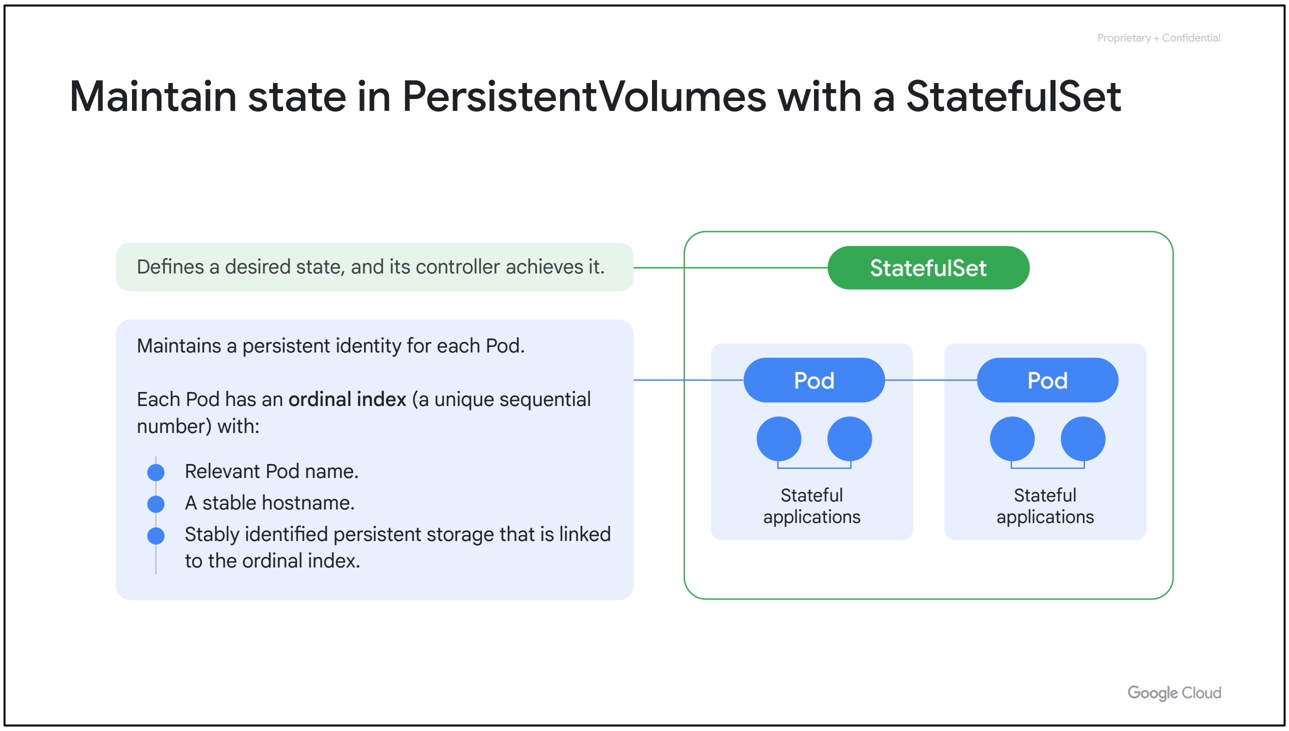
A Pod template used for stateless applications.

Runs and maintains a set of identical Pods, known as replicas.

Replicas need to attach and reattach to PersistentVolumes dynamically, which might cause a deadlock.

Although PersistentVolumes provide durable storage for a Pod, they can also be used for other controllers like Deployments and StatefulSets.

A Deployment is a Pod template, typically used for stateless applications, that runs and maintains a set of identical Pods, commonly known as replicas. These replicas need to attach and reattach to PersistentVolumes dynamically, which might cause a deadlock.



Maintain state in PersistentVolumes with a StatefulSet

Defines a desired state, and its controller achieves it.

Maintains a persistent identity for each Pod.

Each Pod has an **ordinal index** (a unique sequential number) with:

- Relevant Pod name.
- A stable hostname.
- Stably identified persistent storage that is linked to the ordinal index.

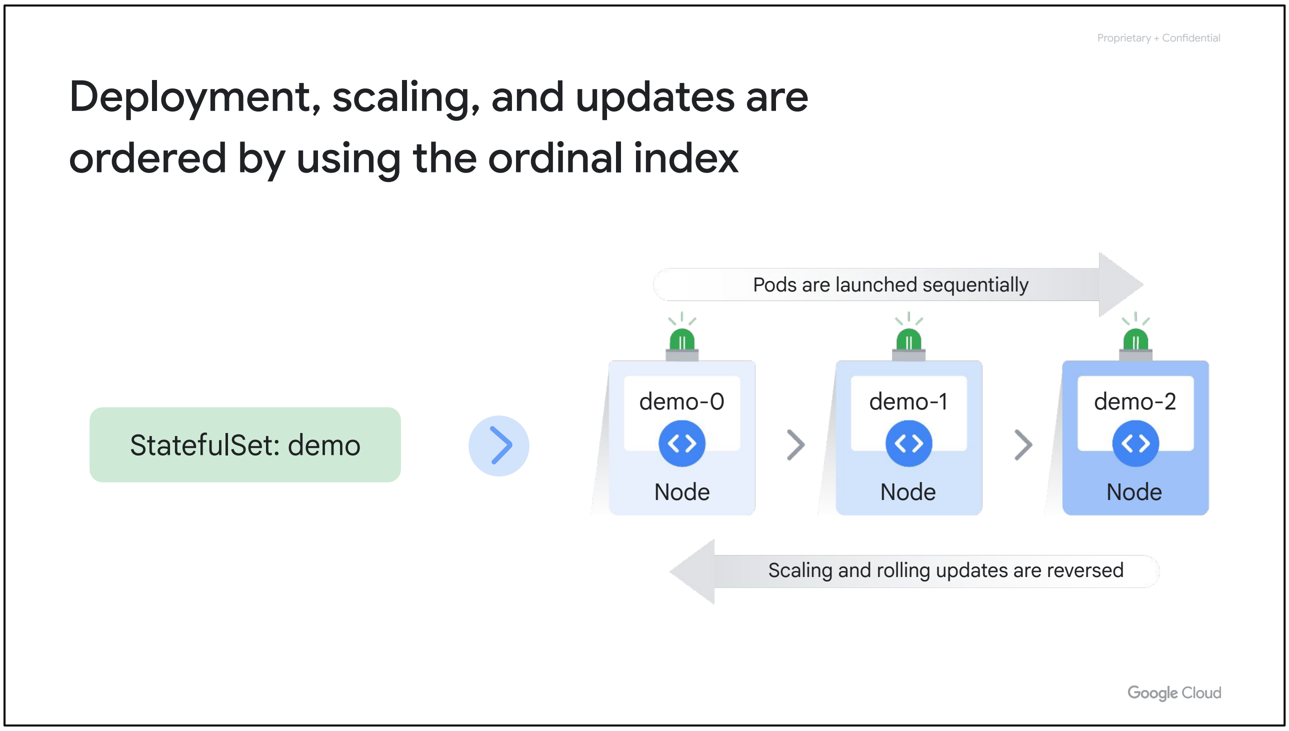
StatefulSet
Pod
Stateful applications
Pod
Stateful applications

StatefulSets help resolve this problem. This means that whenever an application needs to maintain state in PersistentVolumes, it should be managed with a StatefulSet.

Useful for stateful applications, StatefulSets run and maintain a set of Pods. A StatefulSet object defines a desired state, and its controller achieves it. This allows StatefulSets to maintain a persistent identity for each Pod.

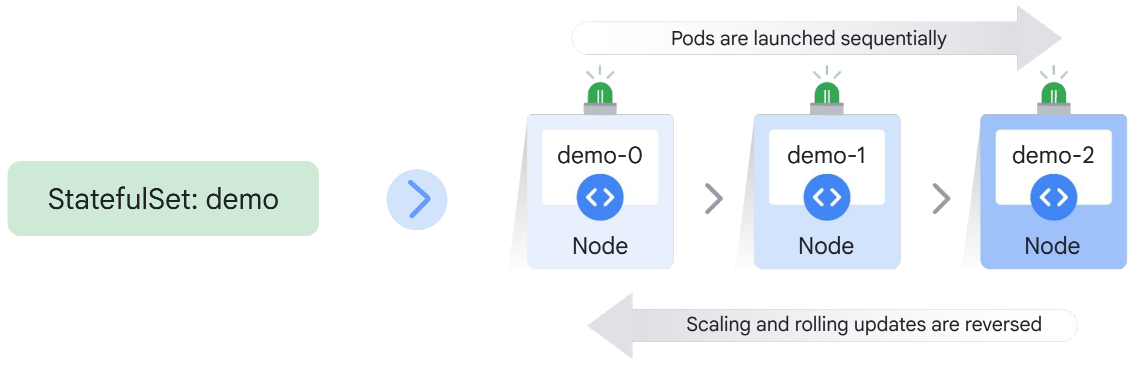
Each Pod in a StatefulSet has an ordinal index with a relevant Pod name, a stable hostname, and stably identified persistent storage that is linked to the ordinal index.

An ordinal index is a unique sequential number that is assigned to each Pod in the StatefulSet. This number defines the Pod’s position in the set’s sequence of Pods.



Proprietary + Confidential

Deployment, scaling, and updates are ordered by using the ordinal index


StatefulSet: demo
Pods are launched sequentially
demo-0
Node
demo-1
Node
demo-2
Node
Scaling and rolling updates are reversed

Google Cloud

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

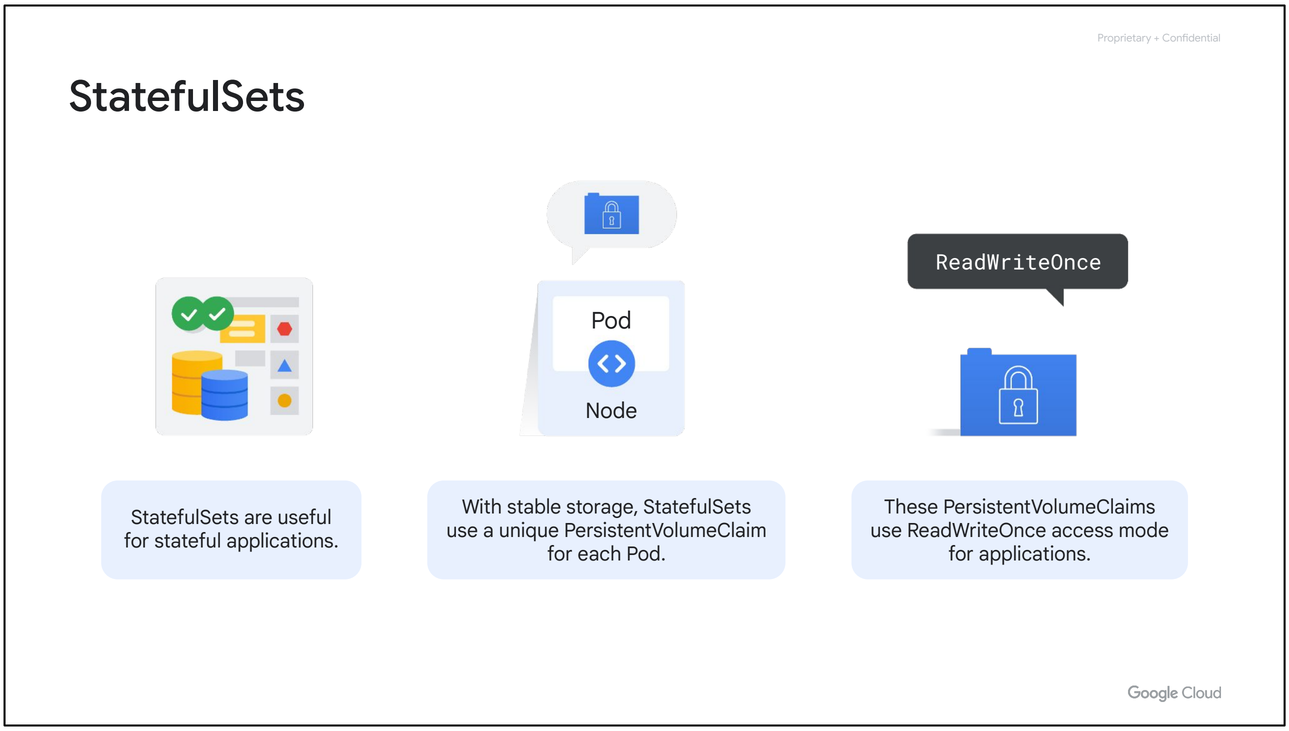
For example, if a StatefulSet named ‘demo’ launches 3 replicas, it will launch Pods named demo-0, demo-1, and demo-2 sequentially.

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 before demo-1.

To launch Pods in parallel without waiting for the Pods to maintain Running and Ready state, ensure that the PodManagementPolicy is Parallel.



StatefulSets

Pod
Node
ReadWriteOnce

StatefulSets are useful for stateful applications.

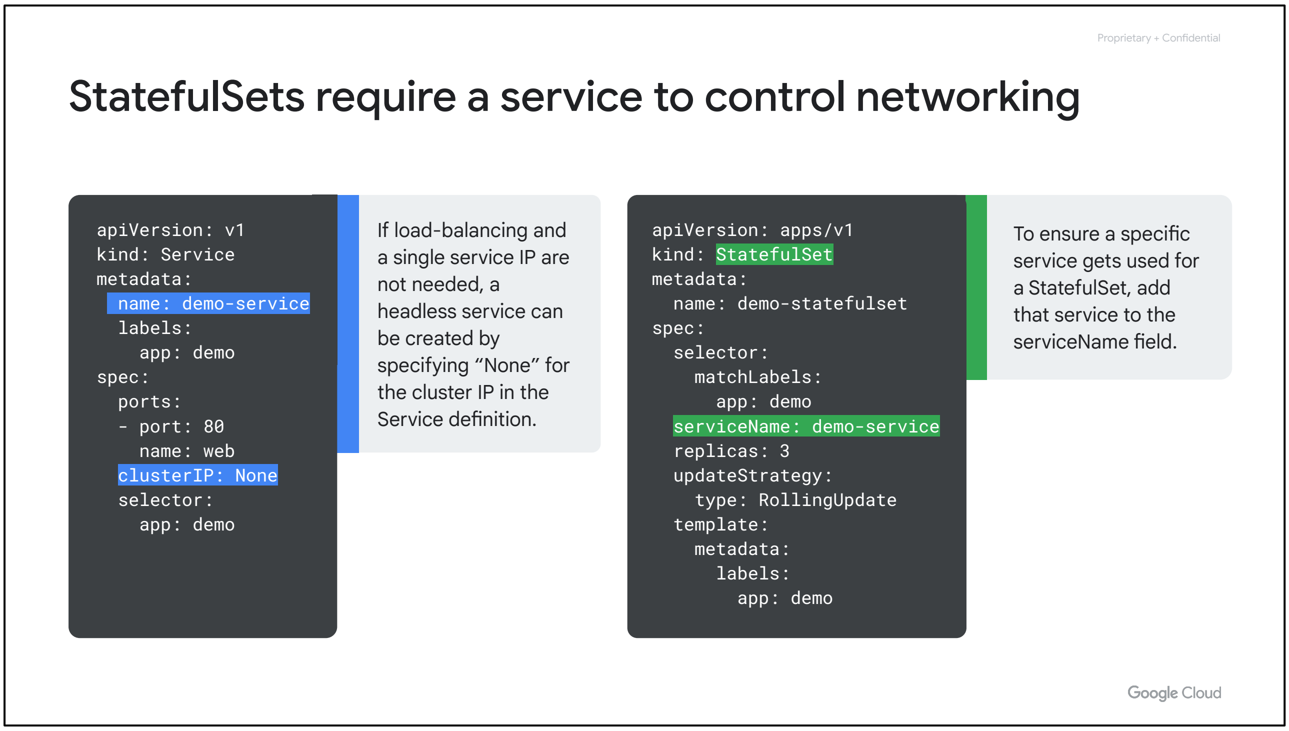
With stable storage, StatefulSets use a unique PersistentVolumeClaim for each Pod.

These PersistentVolumeClaims use ReadWriteOnce access mode for applications.

As mentioned, StatefulSets are useful for stateful applications.

With stable storage, StatefulSets use a unique PersistentVolumeClaim for each Pod. In order for each Pod to 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
```

If load-balancing and a single service IP are not needed, a headless service can be created by specifying “None” for the cluster IP in the Service definition.

```
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
```

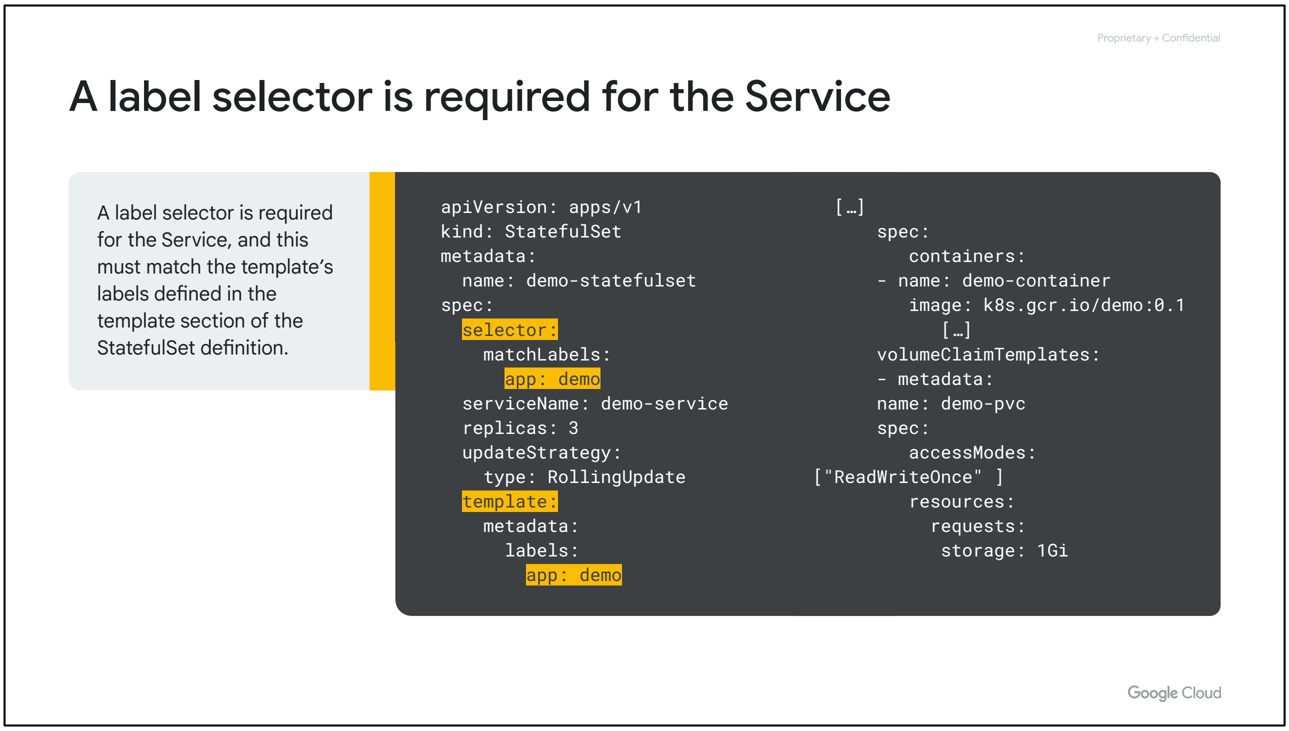
To ensure a specific service gets used for a StatefulSet, add that service to the serviceName field.

Google Cloud

When it comes to controlling networking, StatefulSets require a service.

For example, if load-balancing and a single service IP are not needed, a headless service can be created by specifying “None” for the cluster IP in the Service definition.

To ensure a specific service gets used for a StatefulSet, add that service to the serviceName field.



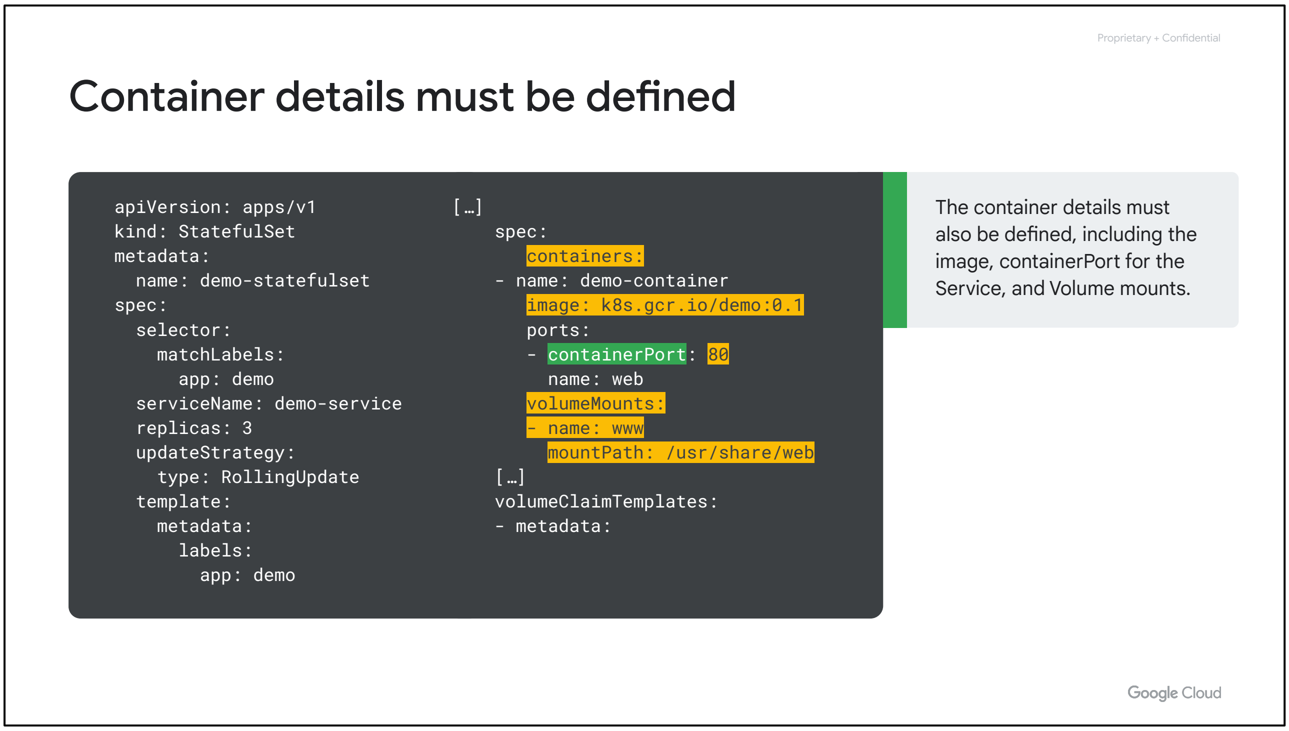
A label selector is required for the Service

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.

```
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
```

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.



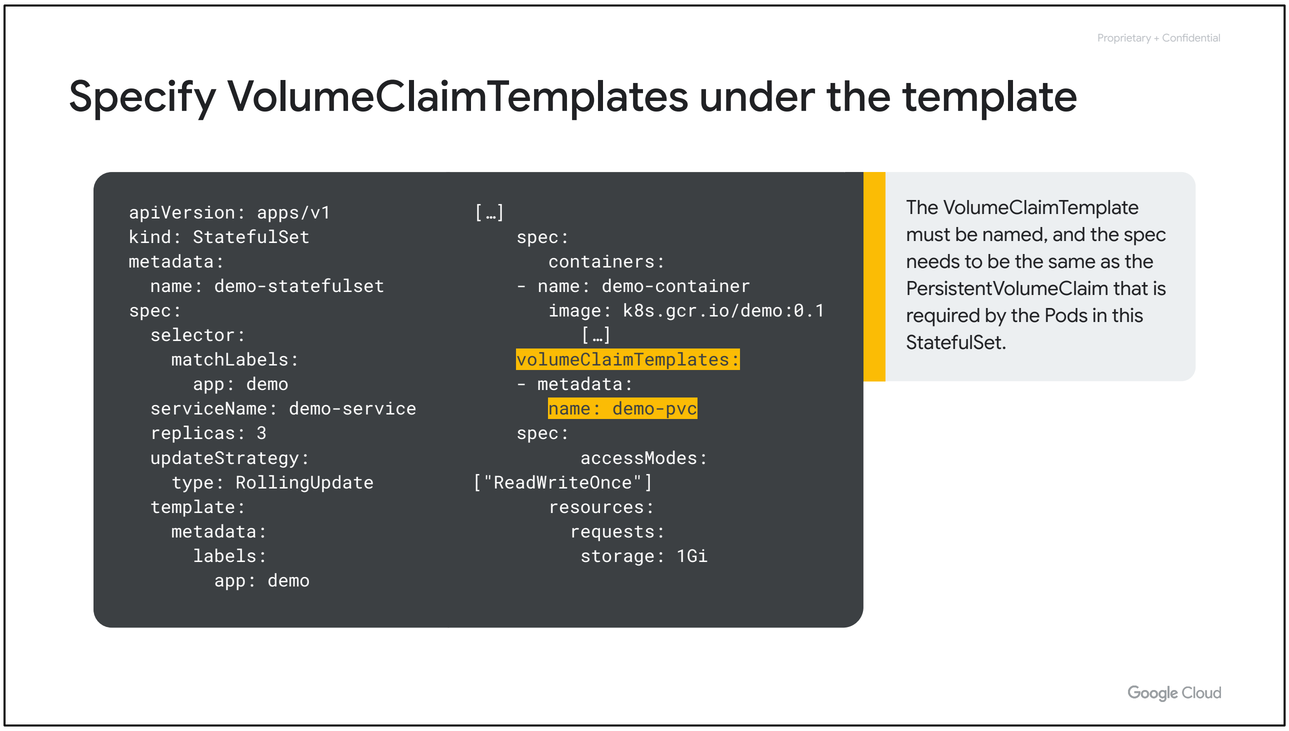
Container details must be defined

```
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:
```

The container details must also be defined, including the image, containerPort for the Service, and Volume mounts.

The container details must also be defined, including the image, containerPort for the Service, and Volume mounts.



Specify VolumeClaimTemplates under the template

```
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
```

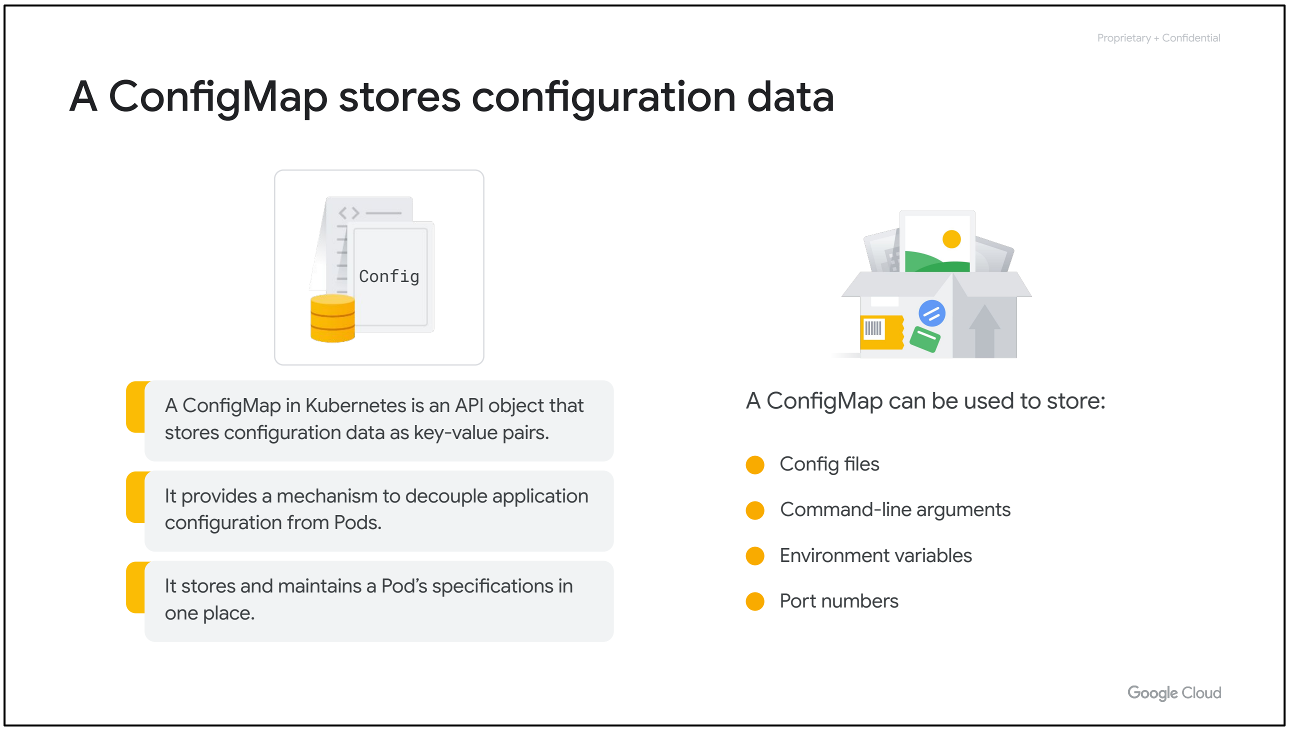
The VolumeClaimTemplate must be named, and the spec needs to be the same as the PersistentVolumeClaim that is required by the Pods in this StatefulSet.

Google Cloud

And most importantly, VolumeClaimTemplates must be specified under the template section.

The VolumeClaimTemplate must be named, and the spec needs to be the same as the PersistentVolumeClaim that is required by the Pods in this StatefulSet.

Persistent Data and Storage



Proprietary + Confidential

A ConfigMap stores configuration data

- A ConfigMap in Kubernetes is an API object that stores configuration data as key-value pairs.
- It provides a mechanism to decouple application configuration from Pods.
- It stores and maintains a Pod's specifications in one place.

A ConfigMap can be used to store:

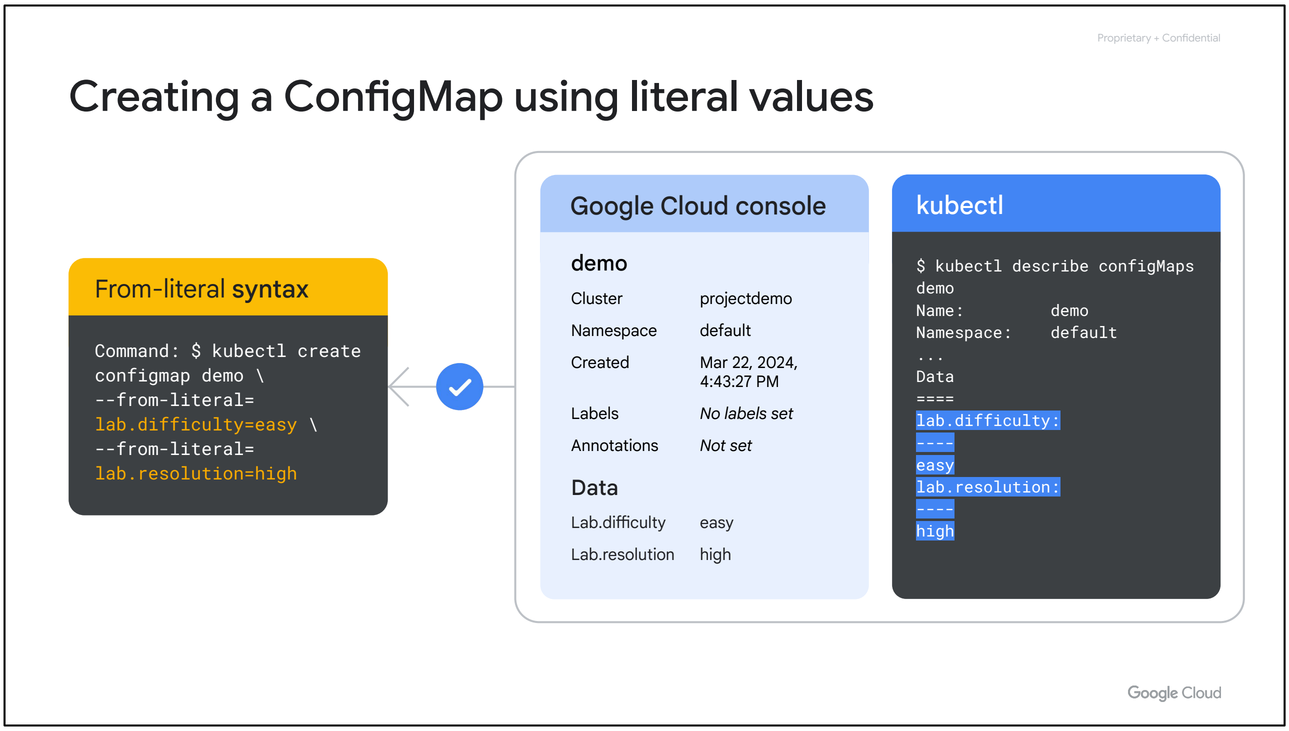
- Config files
- Command-line arguments
- Environment variables
- Port numbers

Google Cloud

A ConfigMap in Kubernetes is an API object that stores configuration data as key-value pairs. It provides a mechanism to decouple application configuration from Pods, which means a Pod's specifications are stored and maintained in one place and act as a single source of truth. This prevents configuration drift.

ConfigMap can be used to store configuration files, command-line arguments, environment variables, port numbers, and other configuration artifacts and make them available inside containers.

This makes applications more portable and manageable without requiring them to be Kubernetes-aware.



Creating a ConfigMap using literal values

From-literal syntax

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

Google Cloud console

demo

Cluster	projectdemo
Namespace	default
Created	Mar 22, 2024, 4:43:27 PM
Labels	*No labels set*
Annotations	*Not set*

Data

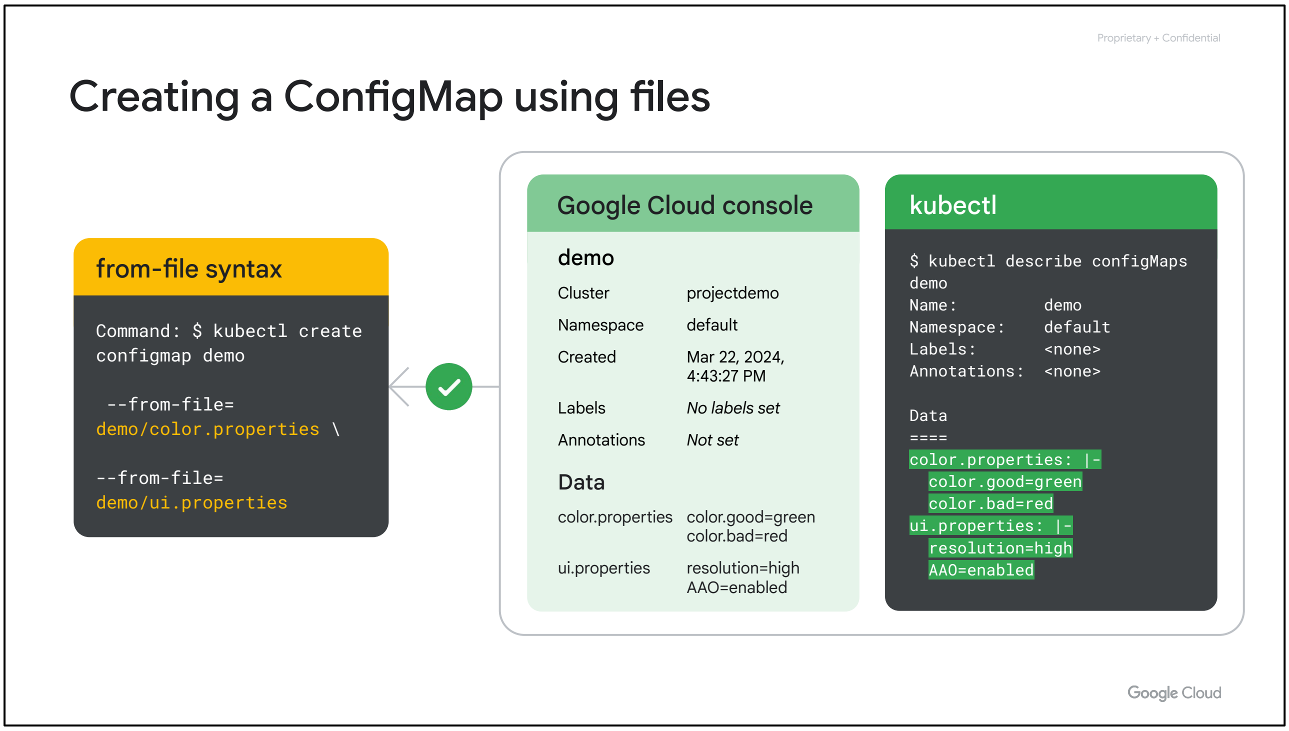
Lab.difficulty	easy
Lab.resolution	high

kubectl

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

Let’s explore an example where a ConfigMap named `demo` is created using literal values.

You can add multiple key-value pairs, such as **lab.difficulty=easy**, and **lab.resolution=high**, and even view details of configMaps using kubectl or the Google Cloud console.



Creating a ConfigMap using files

from-file syntax

```
Command: $ kubectl create
configmap demo

 --from-file=
demo/color.properties \

--from-file=
demo/ui.properties
```

Google Cloud console

demo

Cluster	projectdemo
Namespace	default
Created	Mar 22, 2024, 4:43:27 PM
Labels	*No labels set*
Annotations	*Not set*

Data

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

kubectl

```
$ kubectl describe configMaps
demo
Name:          demo
Namespace:     default
Labels:        <none>
Annotations:   <none>

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

Google Cloud

Another way to create a ConfigMap is by using the `from-file` syntax. These files contain multiple key-values.

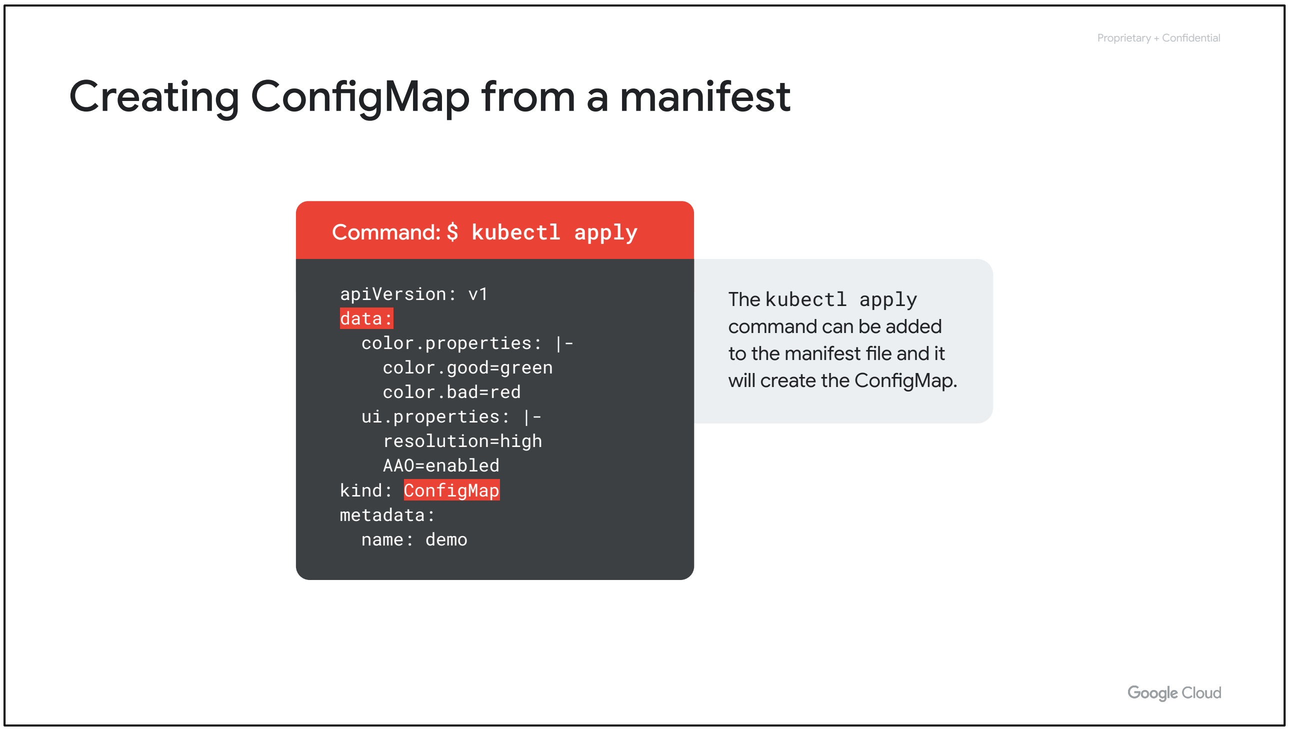
Multiple files can be added to a ConfigMap, and it’s recommended that you check these files into a source code control system to maintain their versioning and history.

Names can be specified for keys. As an alternative to using the source filenames, key names can be added.

This syntax is very similar to the `from-literal` syntax, but here an additional key value is inserted to rename the key used.

Let’s say we have an example where the key value `Color` is added for the file called `color.properties`, and then the key value `Graphics` is added for the file called `ui.properties`.

The contents of ConfigMaps can be verified using kubectl or the Google Cloud console.


Proprietary + Confidential

Creating ConfigMap from a manifest

Command: `$ kubectl apply`

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

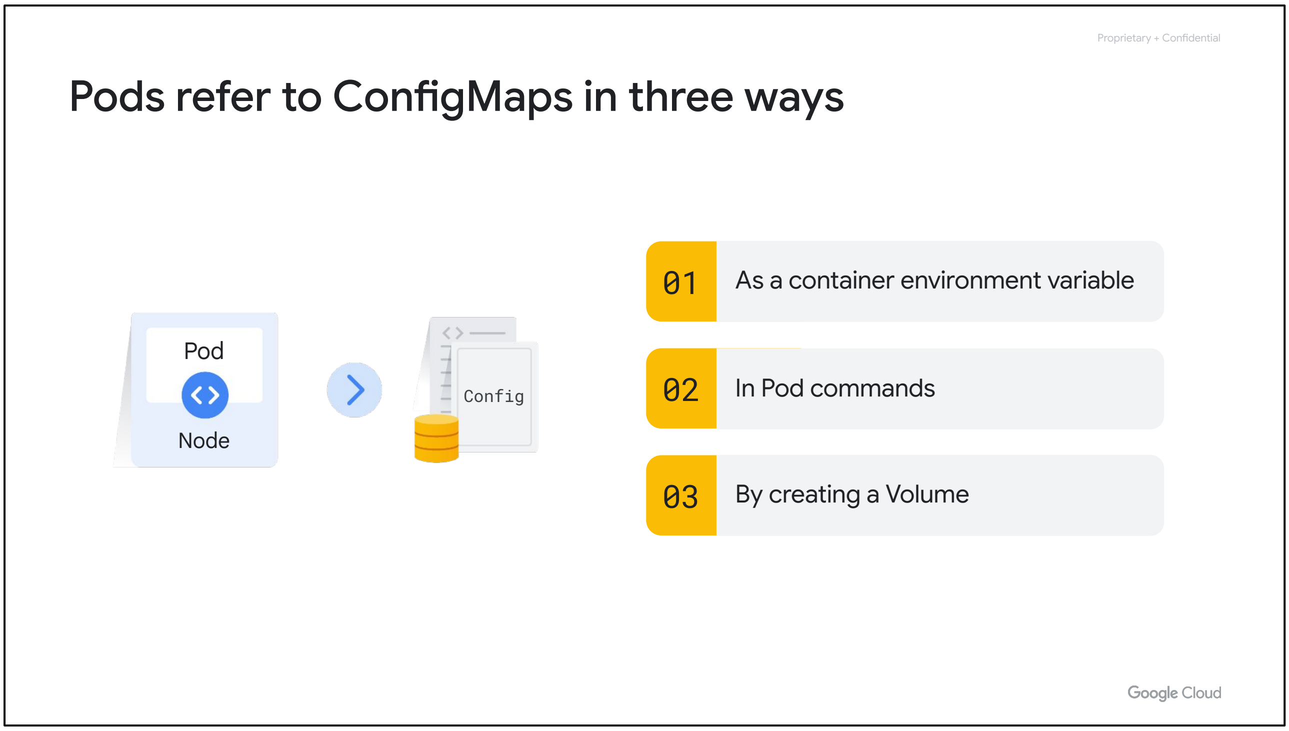
The `kubectl apply` command can be added to the manifest file and it will create the ConfigMap.

Google Cloud

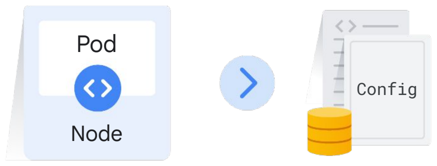
ConfigMaps can also be created from a manifest.

The data is the same as in the previous examples.

The `kubectl apply` command can be added to the manifest file and it will create the ConfigMap.



Pods refer to ConfigMaps in three ways

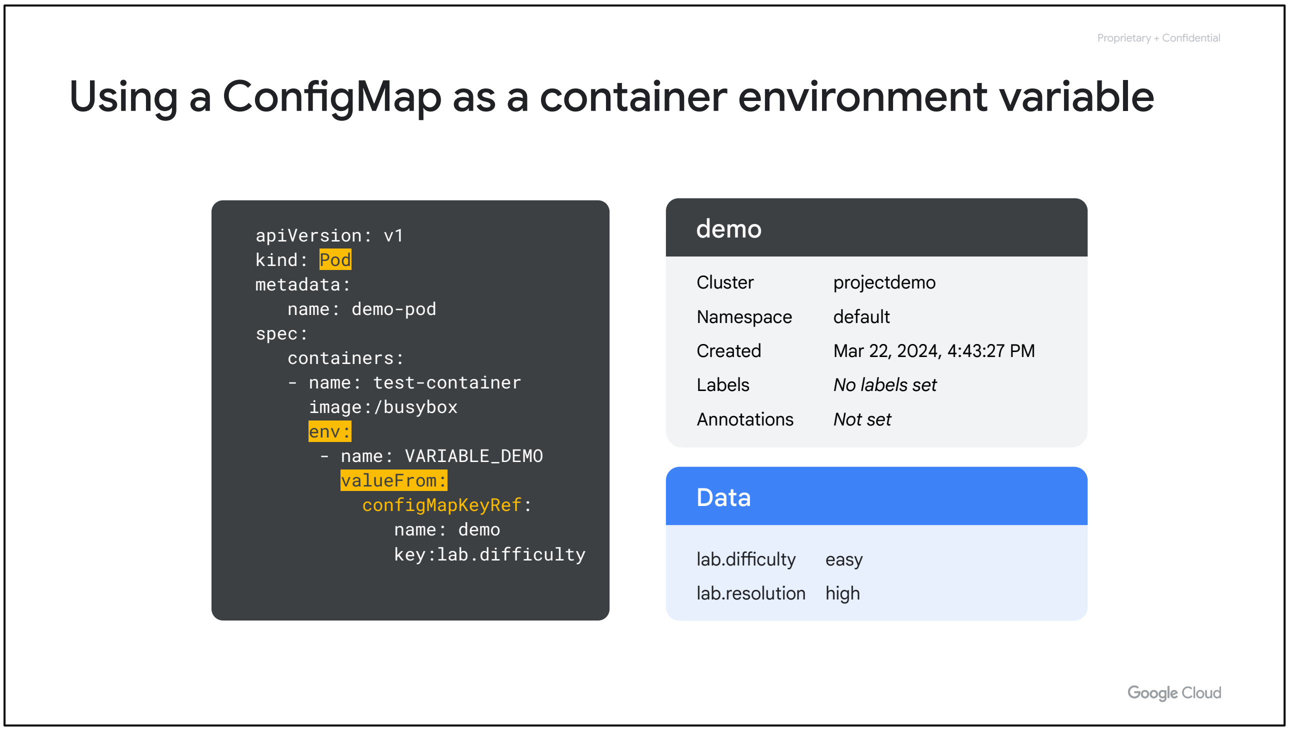


01 As a container environment variable

02 In Pod commands

03 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	Mar 22, 2024, 4:43:27 PM
Labels	*No labels set*
Annotations	*Not set*

Data

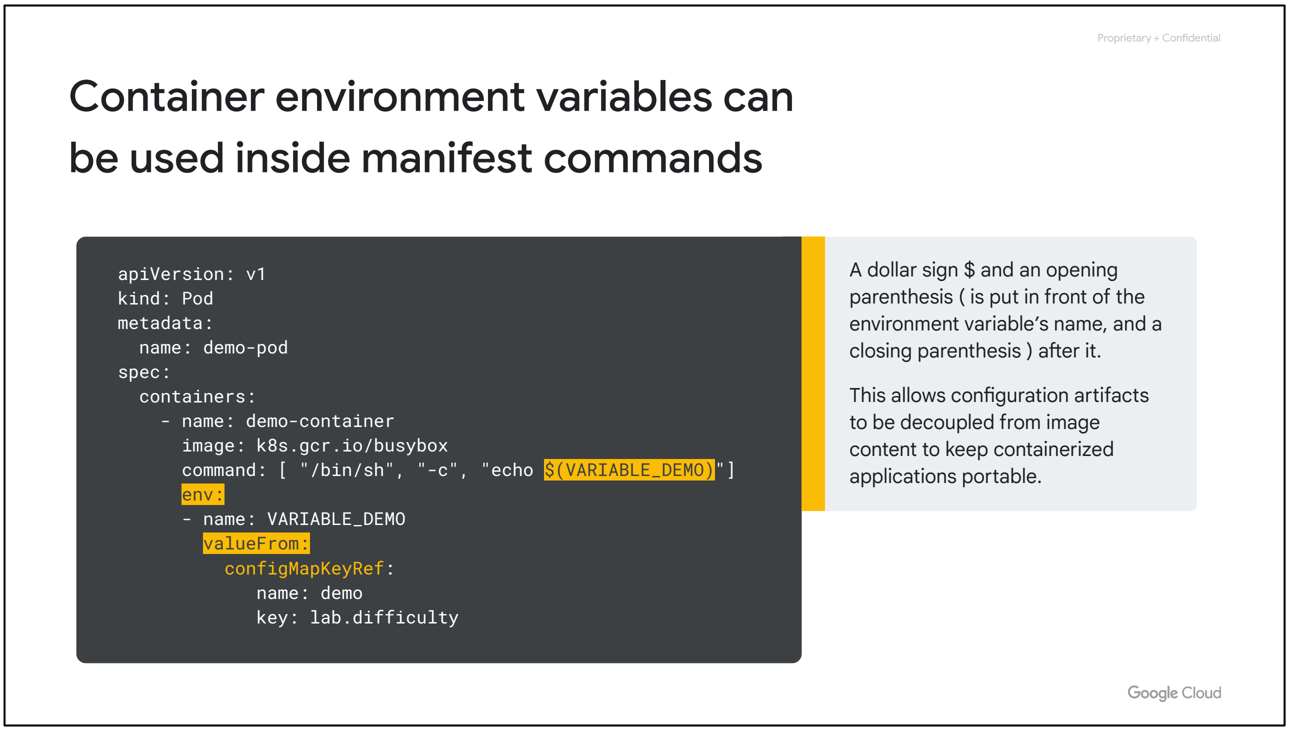
lab.difficulty	easy
lab.resolution	high

For example, let's say we have a single ConfigMap that is used in the Pod as a container environment variable.

Within an `env` field in the YAML file, a container environment variable can be named as `VARIABLE_DEMO`.

The values are retrieved using `configMapKeyRef`.

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



Container environment variables can be used inside manifest 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
```

A dollar sign $ and an opening parenthesis (is put in front of the environment variable's name, and a closing parenthesis) after it.

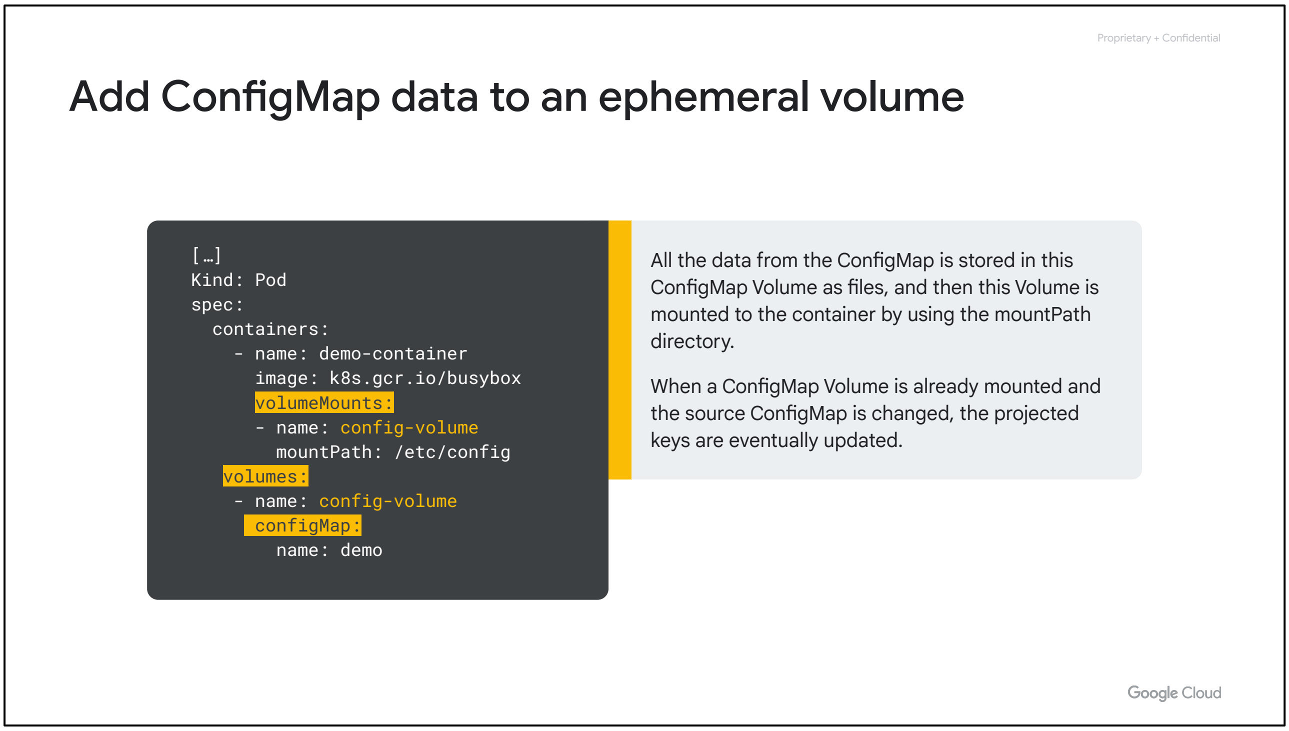
This allows configuration artifacts to be decoupled from image content to keep containerized applications portable.

Google Cloud

After the container environmental variables are defined, they can be used inside Pod manifest commands.

A dollar sign `$` and an opening parenthesis `(` is put in front of the environment variable's name, and a closing parenthesis `)` after it. This allows configuration artifacts to be decoupled from image content to keep containerized applications portable.

As a result, the kubelet has no way to reach into the Pod and modify these values later.



Add ConfigMap data to an ephemeral 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
```

All the data from the ConfigMap is stored in this ConfigMap Volume as files, and then this Volume is mounted to the container by using the mountPath directory.

When a ConfigMap Volume is already mounted and the source ConfigMap is changed, the projected keys are eventually updated.

ConfigMap data can also be added into an ephemeral Volume.

For example, let’s say a Volume named `config-volume` is created in the Volumes section, with a ConfigMap named demo.

The result is that a ConfigMap Volume is created for this Pod. This means that all the data from the ConfigMap is stored in this ConfigMap Volume as files, and then this Volume is mounted to the container by using the `mountPath` directory.

When a ConfigMap Volume is already mounted and the source ConfigMap is changed, the projected keys are eventually updated.

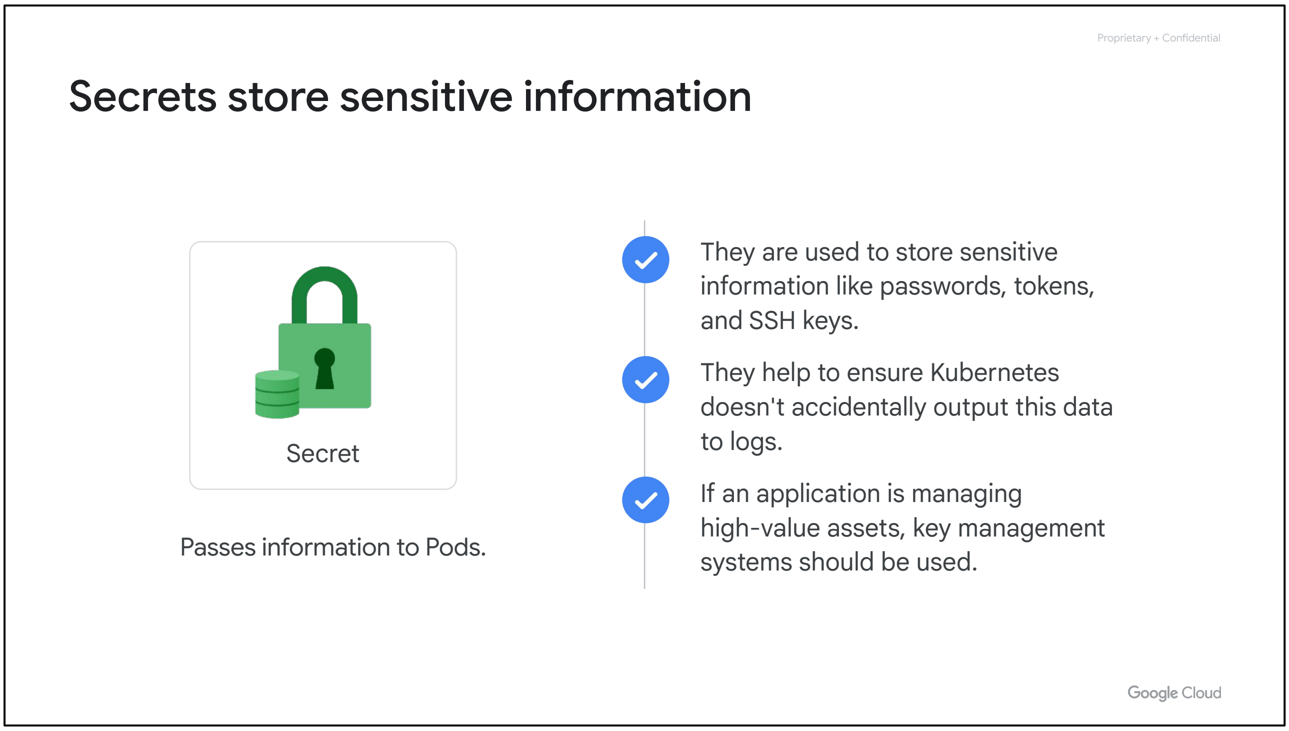
Persistent Data and Storage

01 Kubernetes storage abstractions

02 StatefulSets

03 Configmaps

04 Secrets



Secrets store sensitive information

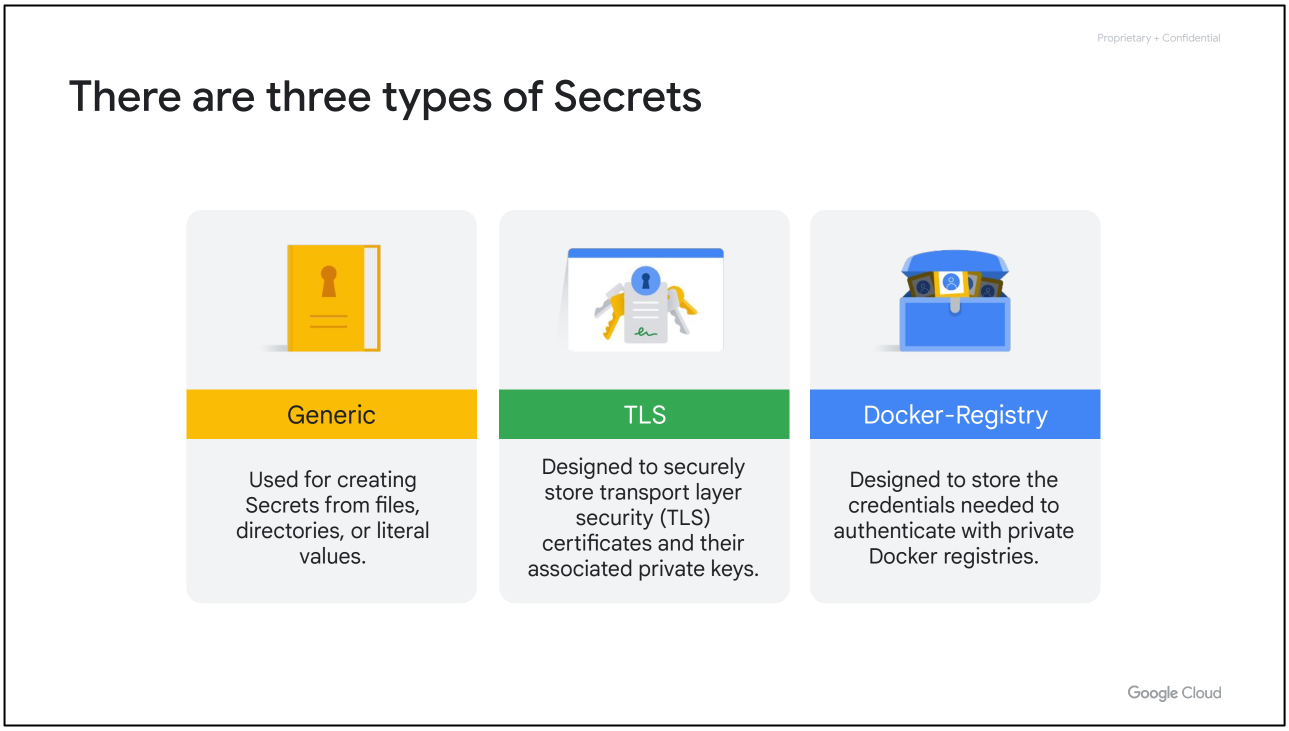
Passes information to Pods.

- ✓ They are used to store sensitive information like passwords, tokens, and SSH keys.
- ✓ They help to ensure Kubernetes doesn't accidentally output this data to logs.
- ✓ If an application is managing high-value assets, key management systems should be used.

Similar to ConfigMaps, Secrets pass information to Pods. However, the difference is that Kubernetes applications use Secrets to store sensitive information like passwords, tokens, and SSH keys. This lets users manage sensitive information in their own control plane.

Secrets also help to ensure Kubernetes doesn't accidentally output this data to logs.

That being said, if an application is managing high-value assets or requires stringent regulatory requirements, key management systems like Cloud Key Management Service, also referred to as Cloud KMS, should be used for full secret management.



There are three types of Secrets

Generic

Used for creating Secrets from files, directories, or literal values.

TLS

Designed to securely store transport layer security (TLS) certificates and their associated private keys.

Docker-Registry

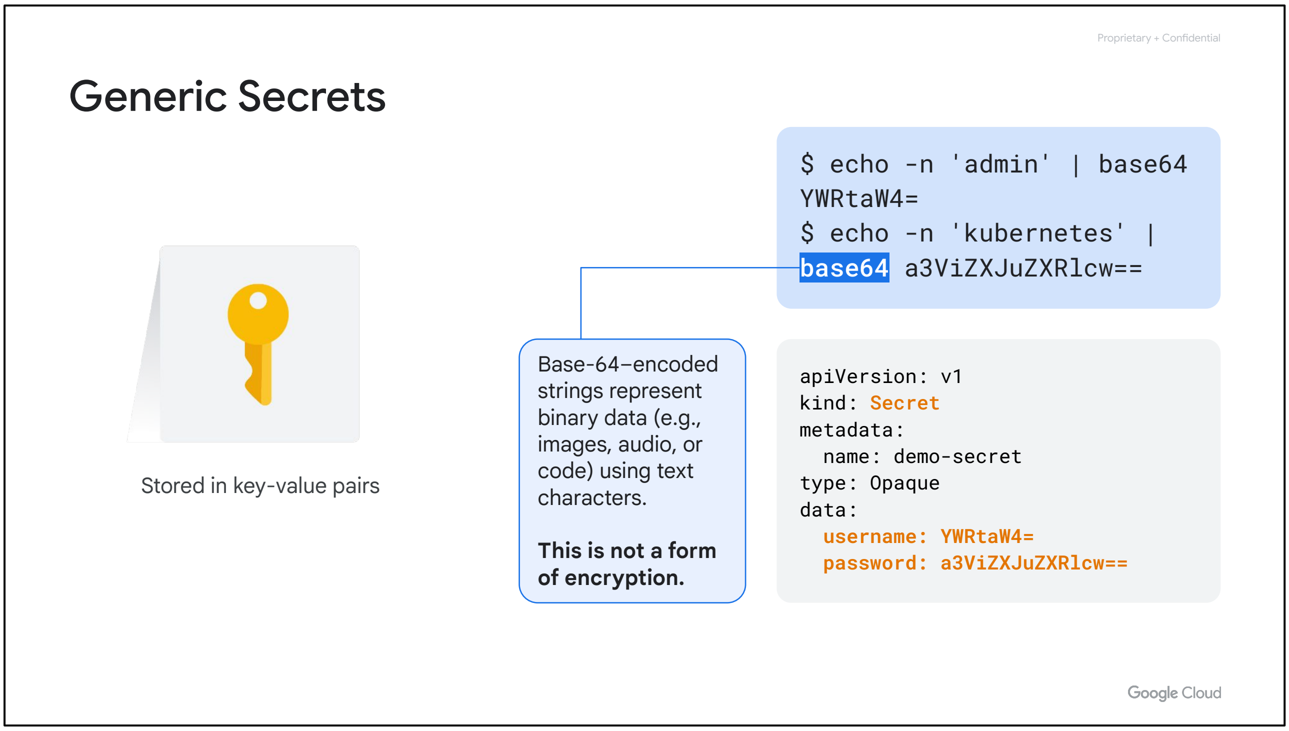
Designed to store the credentials needed to authenticate with private Docker registries.

There are three types of Secrets: generic, TLS, and Docker-Registry.

Generic Secrets are used for creating Secrets from files, directories, or literal values.

TLS Secrets are designed to securely store transport layer security (TLS) certificates and their associated private keys. These are the digital credentials used to establish encrypted communication between different components in a cluster.

And **Docker-Registry Secrets** are designed to store the credentials needed to authenticate with private Docker registries. However, in GKE, Artifact Registry integrates with Identity and Access Management, so this Secret type is not needed.



Generic Secrets

Stored in key-value pairs

```
$ echo -n 'admin' | base64
YWRtaW4=
$ echo -n 'kubernetes' |
base64 a3ViZXJuZXRlcw==
```

Base-64–encoded strings represent binary data (e.g., images, audio, or code) using text characters.

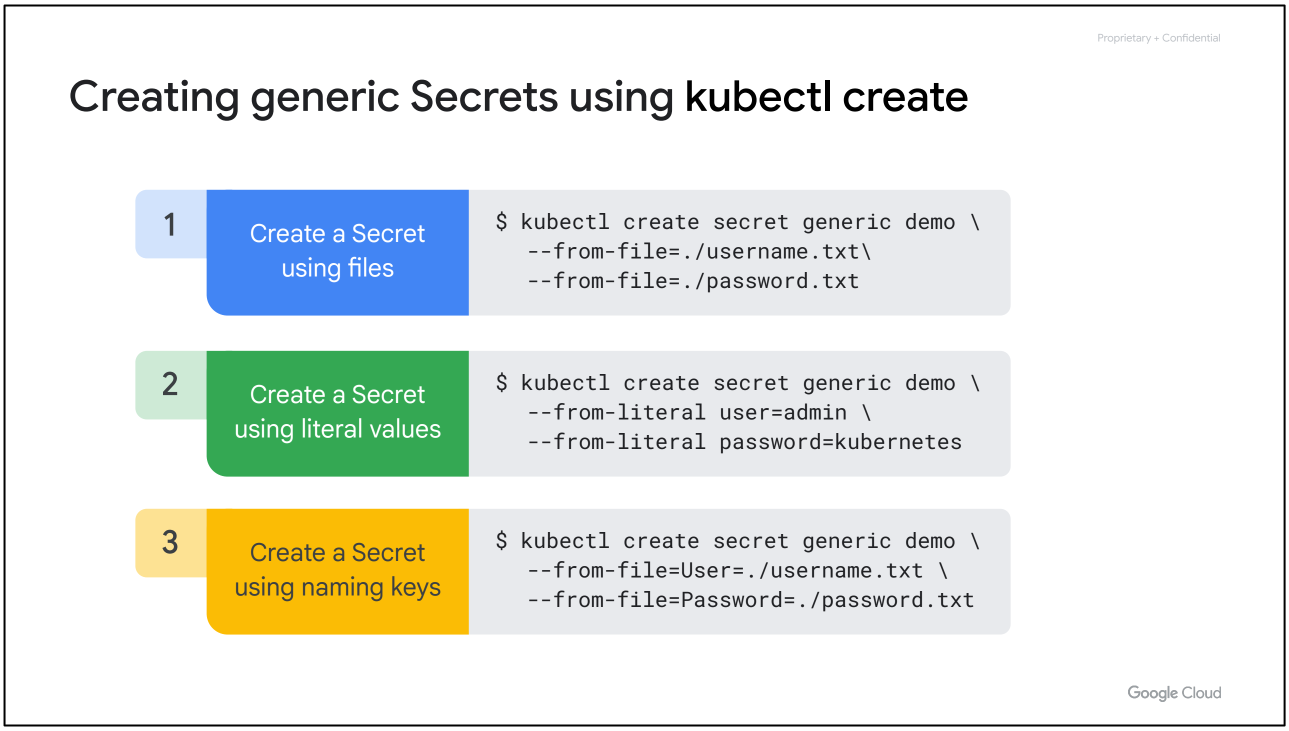
This is not a form of encryption.

```
apiVersion: v1
kind: Secret
metadata:
  name: demo-secret
type: Opaque
data:
  username: YWRtaW4=
  password: a3ViZXJuZXRlcw==
```

Let’s explore the generic Secret type in some more detail.

Just like ConfigMaps, generic Secrets are stored in key-value pairs. However, in Secrets, values are base-64–encoded strings. Base-64 encoding is a way of representing binary data–like images, audio, or code–using text characters.

Now it’s important to note that base-64 encoding is not a form of encryption. If encryption is required, please use an encryption key management system like Cloud KMS. Any encoded strings produced can then be used in the Secret manifest.



Proprietary + Confidential

Creating generic Secrets using **kubectl create**

1 Create a Secret using files

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

2 Create a Secret using literal values

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

3 Create a Secret using naming keys

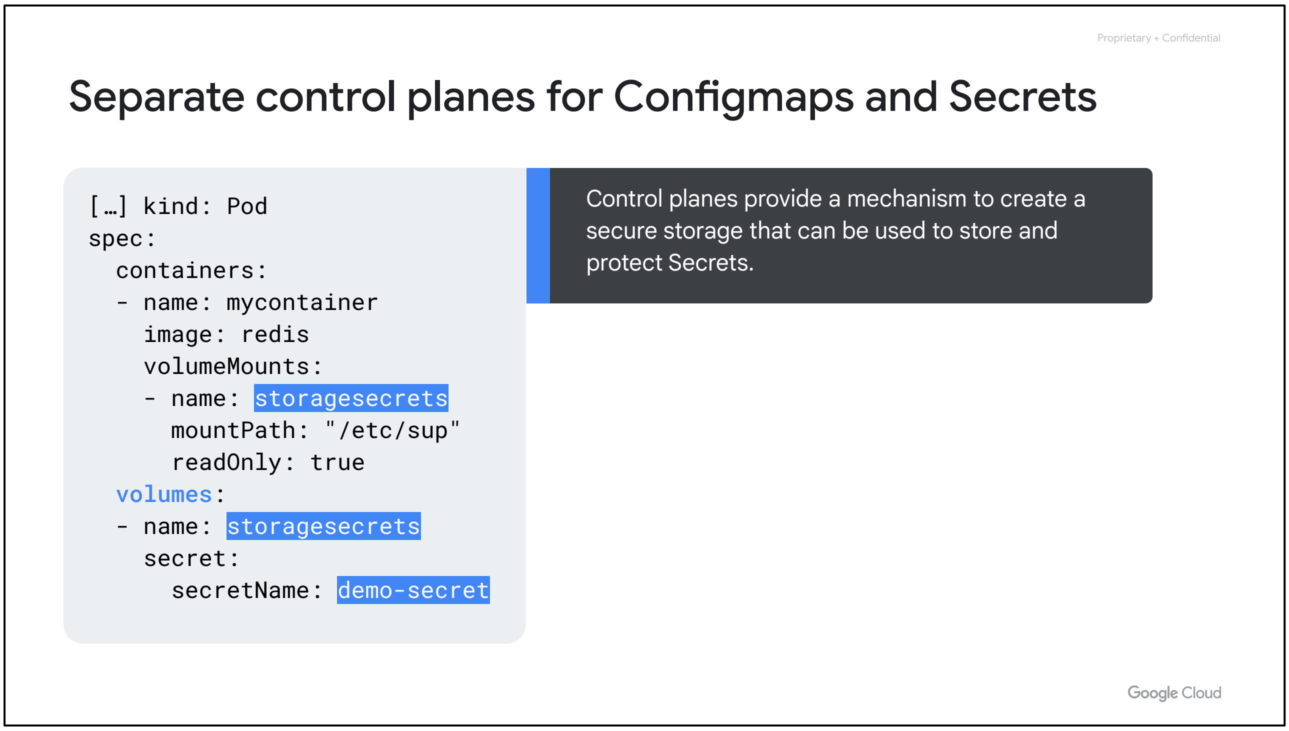
```
$ kubectl create secret generic demo \
   --from-file=User=./username.txt \
   --from-file=Password=./password.txt
```

Google Cloud

But how exactly do you create a Secret? The answer is by using the `kubectl create secret` command.

1. One option is to create a Secret using files, which means populating the Secret object with data sourced directly from files on your local system.
2. A second option is to create a Secret using literal values, which means defining the secret data directly within the Secret manifest file itself.
3. And a third option is to create a Secret using naming keys, which means using the default key name as the file name.

For all three options, the syntax is similar. The differences come from the Secrets entry details.



Separate control planes for Configmaps and Secrets

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

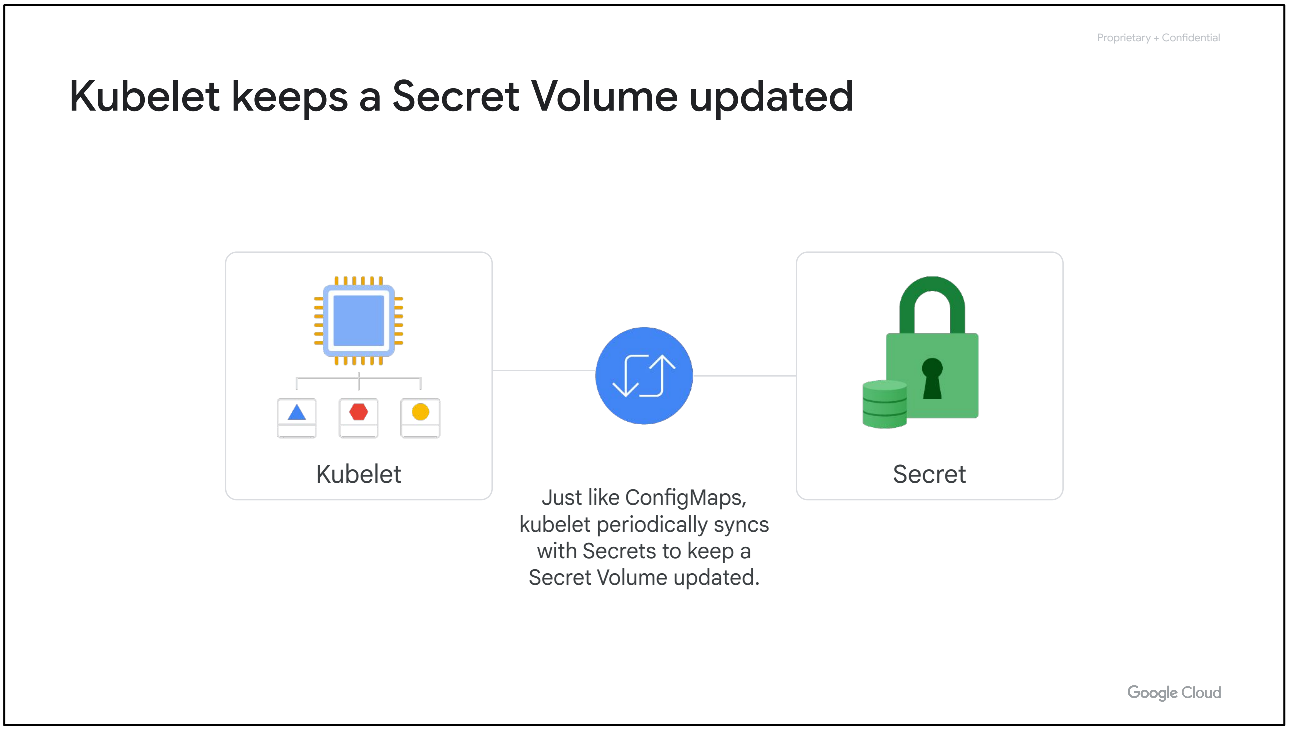
Control planes provide a mechanism to create a secure storage that can be used to store and protect Secrets.

There are separate control planes for Configmaps and Secrets. Control planes provide a mechanism to create a secure storage that can be used to store and protect Secrets.

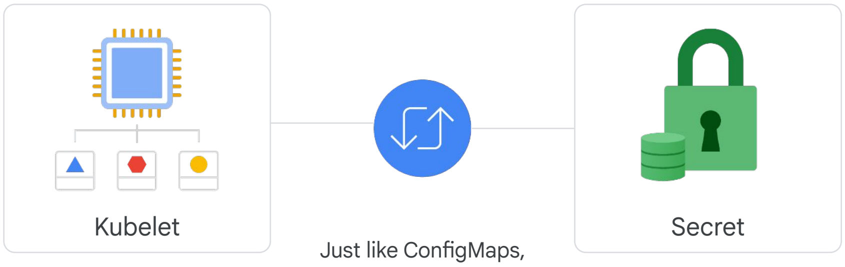
Let's say a Secret Volume named `storagesecrets` is created and refers to a Secret named `demo-secret`.

This Volume is mounted to the container with read-only access. This Volume can be used by multiple containers within the Pod. Which allows for better latency, because pods do not need to wait to gain access to the secret.

A password key for the Secret will not be displayed. If a password key is required, it must be listed under the items field.



Kubelet keeps a Secret Volume updated



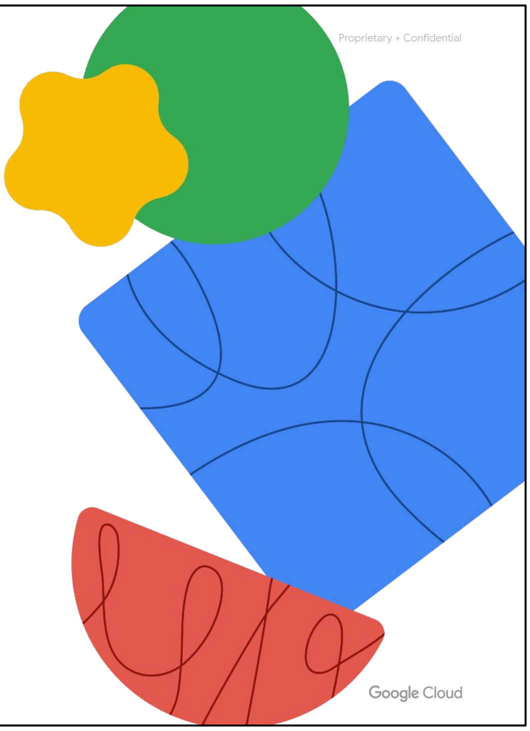
Just like ConfigMaps, kubelet periodically syncs with Secrets to keep a Secret Volume updated.

Just like ConfigMaps, kubelet periodically syncs with Secrets to keep a Secret Volume updated.

If a Secret that is already attached as a Volume has changed, the keys and values will eventually be updated.

Quiz questions

Let’s pause for a quick check in.



Quiz | Question 1

Question

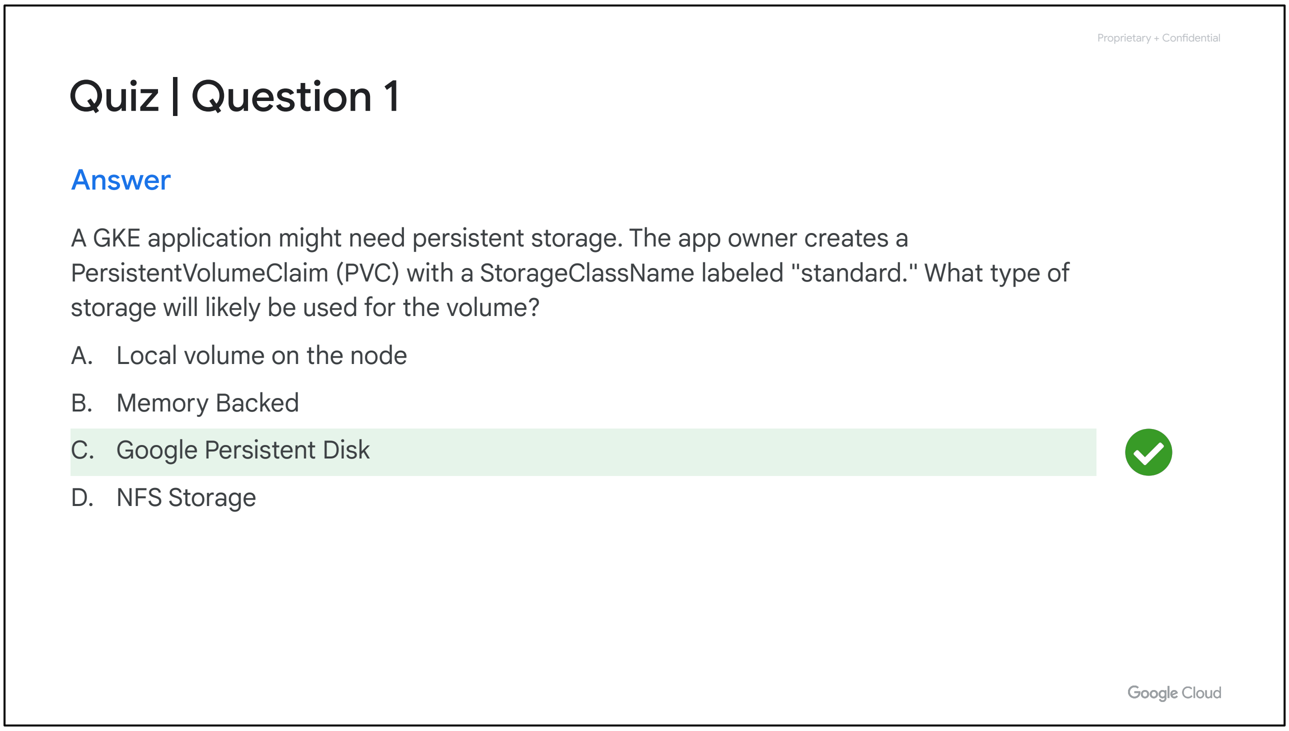
A GKE application might need persistent storage. The app owner creates a PersistentVolumeClaim (PVC) with a StorageClassName labeled "standard." What type of storage will likely be used for the volume?

A. Local volume on the node

B. Memory Backed

C. Google Persistent Disk

D. NFS Storage



Quiz | Question 1

Answer

A GKE application might need persistent storage. The app owner creates a PersistentVolumeClaim (PVC) with a StorageClassName labeled "standard." What type of storage will likely be used for the volume?

A. Local volume on the node

B. Memory Backed

C. Google Persistent Disk ✓

D. NFS Storage

A GKE application might need persistent storage. The app owner creates a PersistentVolumeClaim (PVC) with a StorageClassName labeled "standard." What type of storage will likely be used for the volume?

A. Local volume on the node: This is the **incorrect answer** because **local volumes** are tied to a specific node and don't provide the persistent storage needed if a pod is moved or the node fails..
B. Memory Backed: This is the **incorrect answer** because **memory-backed volumes** are ephemeral and lose data when the pod restarts. They are not suitable for persistent storage..
C. Google Persistent Disk: This is the **correct answer** because **Google Persistent Disk,**provides persistent storage that is independent of the lifecycle of individual nodes, ensuring data availability even if pods are rescheduled or nodes fail..
D. NFS Storage: This is the **incorrect answer** because while NFS can be used for persistent storage in Kubernetes, the "standard" StorageClassName in GKE specifically refers to Google Persistent Disk..

Quiz | Question 2

Question

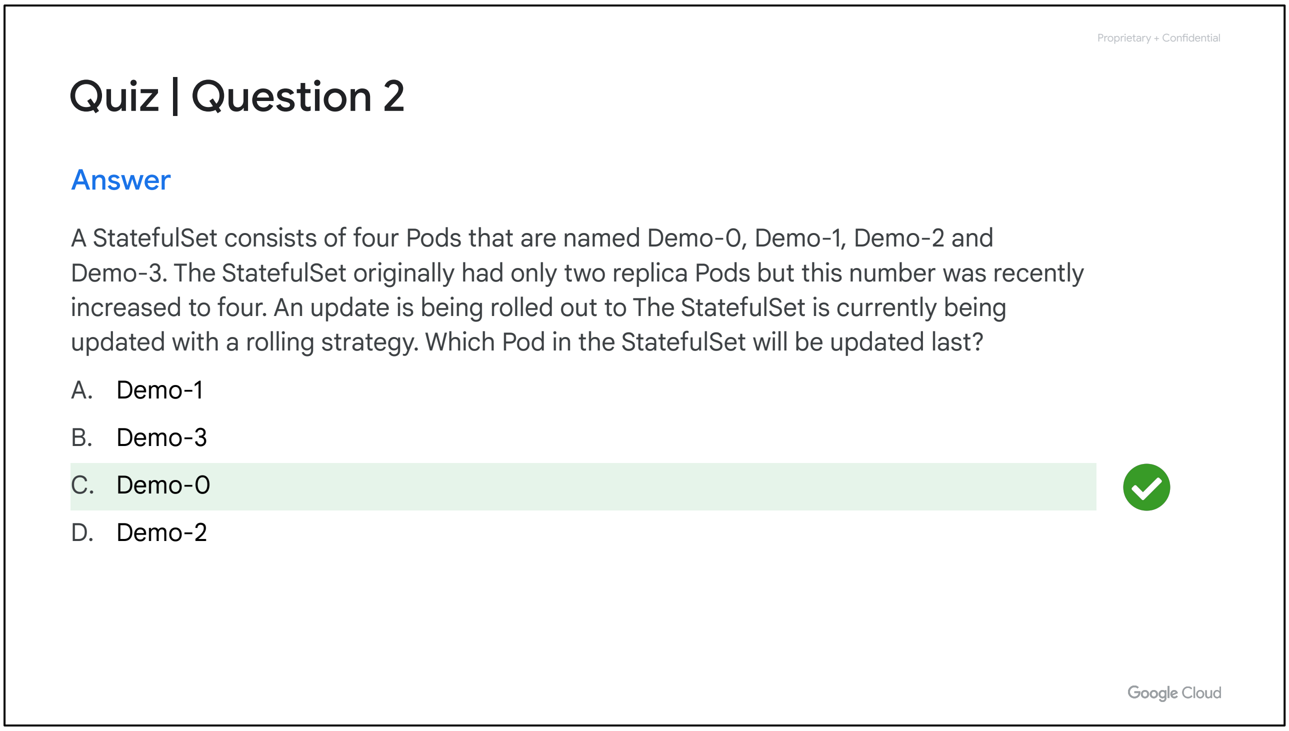
A StatefulSet consists of four Pods that are named Demo-0, Demo-1, Demo-2 and Demo-3. The StatefulSet originally had only two replica Pods but this number was recently increased to four. An update is being rolled out to The StatefulSet is currently being updated with a rolling strategy. Which Pod in the StatefulSet will be updated last?

A. Demo-1

B. Demo-3

C. Demo-0

D. Demo-2



Quiz | Question 2

Answer

A StatefulSet consists of four Pods that are named Demo-0, Demo-1, Demo-2 and Demo-3. The StatefulSet originally had only two replica Pods but this number was recently increased to four. An update is being rolled out to The StatefulSet is currently being updated with a rolling strategy. Which Pod in the StatefulSet will be updated last?

A. Demo-1

B. Demo-3

C. Demo-0 ✓

D. Demo-2

A StatefulSet consists of four Pods that are named Demo-0, Demo-1, Demo-2 and Demo-3. The StatefulSet originally had only two replica Pods but this number was recently increased to four. An update is being rolled out to The StatefulSet is currently being updated with a rolling strategy. Which Pod in the StatefulSet will be updated last?

A. Demo-1: This is the **incorrect answer** because scaling and rolling updates happen in reverse order.
B. Demo-3: This is the **incorrect answer** because scaling and rolling updates happen in reverse order.
C. Demo-0: This is the **correct answer** because scaling and rolling updates happen in reverse order.
D. Demo-2: This is the **incorrect answer** because scaling and rolling updates happen in reverse order.

Quiz | Question 3

Question

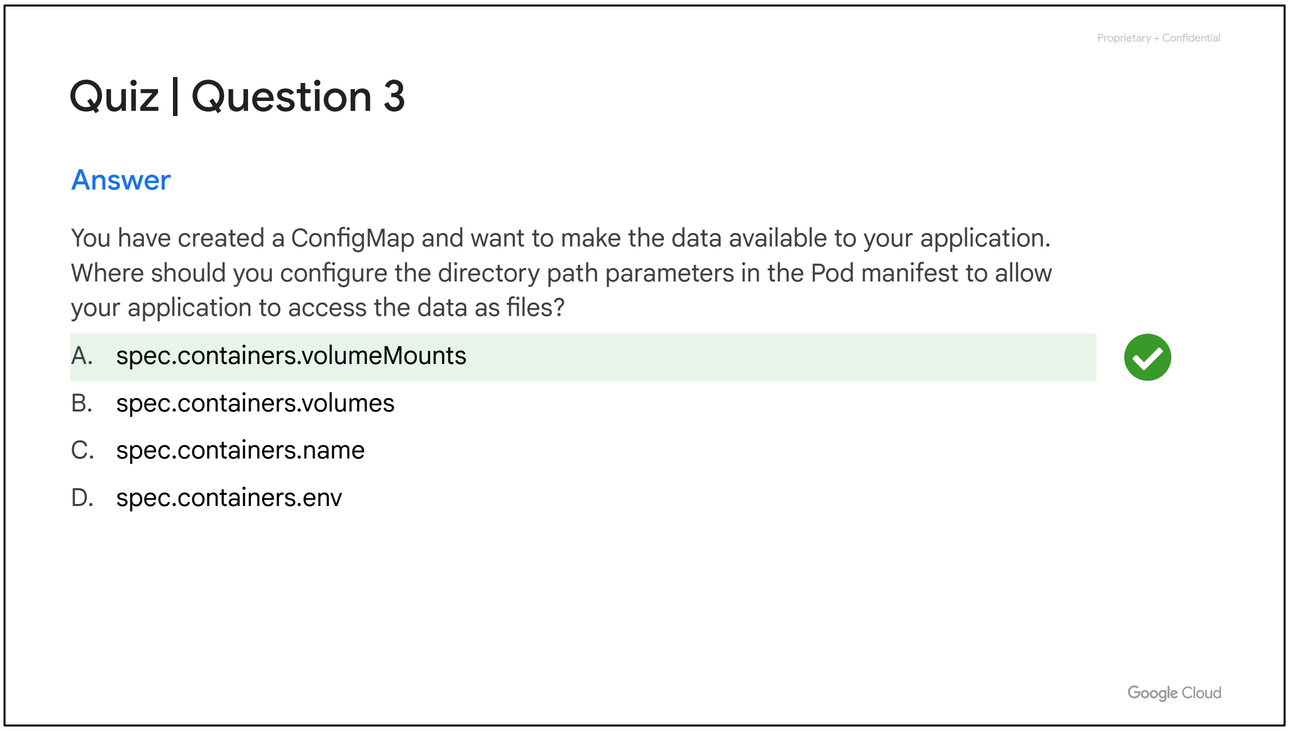
You have created a ConfigMap and want to make the data available to your application. Where should you configure the directory path parameters in the Pod manifest to allow your application to access the data as files?

A. spec.containers.volumeMounts

B. spec.containers.volumes

C. spec.containers.name

D. spec.containers.env



Quiz | Question 3

Answer

You have created a ConfigMap and want to make the data available to your application. Where should you configure the directory path parameters in the Pod manifest to allow your application to access the data as files?

A. spec.containers.volumeMounts ✓
B. spec.containers.volumes
C. spec.containers.name
D. spec.containers.env

You have created a ConfigMap and want to make the data available to your application. Where should you configure the directory path parameters in the Pod manifest to allow your application to access the data as files?

A. spec.containers.volumeMounts: This is the **correct answer** because this is where you specify the directory within a container where a volume should be mounted.
B. spec.containers.volumes: This is the **incorrect answer** because you wouldn't configure the directory path parameters in this section..
C. spec.containers.name: This is the **incorrect answer** because this field is used to specify the name of a container within a Pod, not for configuring volume mount paths.
D. spec.containers.env: This is the **incorrect answer** because this field is used to set environment variables within a container.

Quiz | Question 4

Question

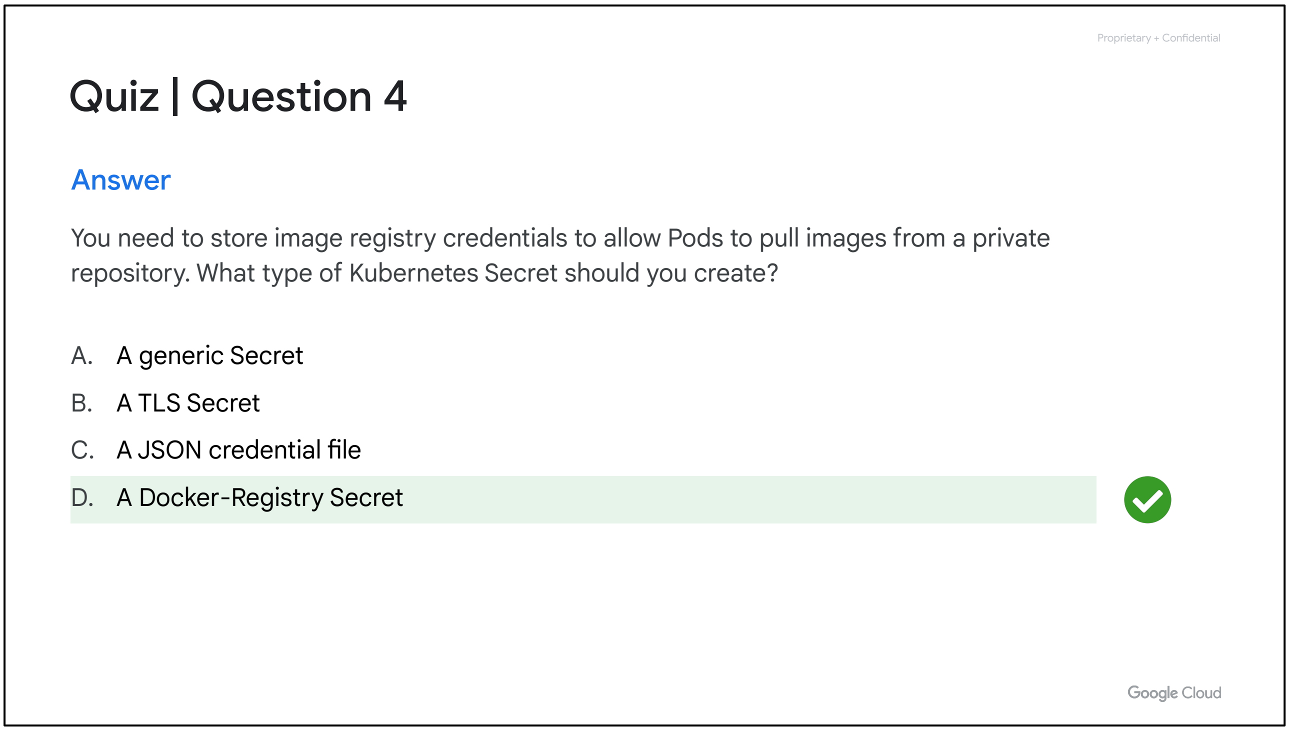
You need to store image registry credentials to allow Pods to pull images from a private repository. What type of Kubernetes Secret should you create?

A. A generic Secret

B. A TLS Secret

C. A JSON credential file

D. A Docker-Registry Secret



Quiz | Question 4

Answer

You need to store image registry credentials to allow Pods to pull images from a private repository. What type of Kubernetes Secret should you create?

A. A generic Secret

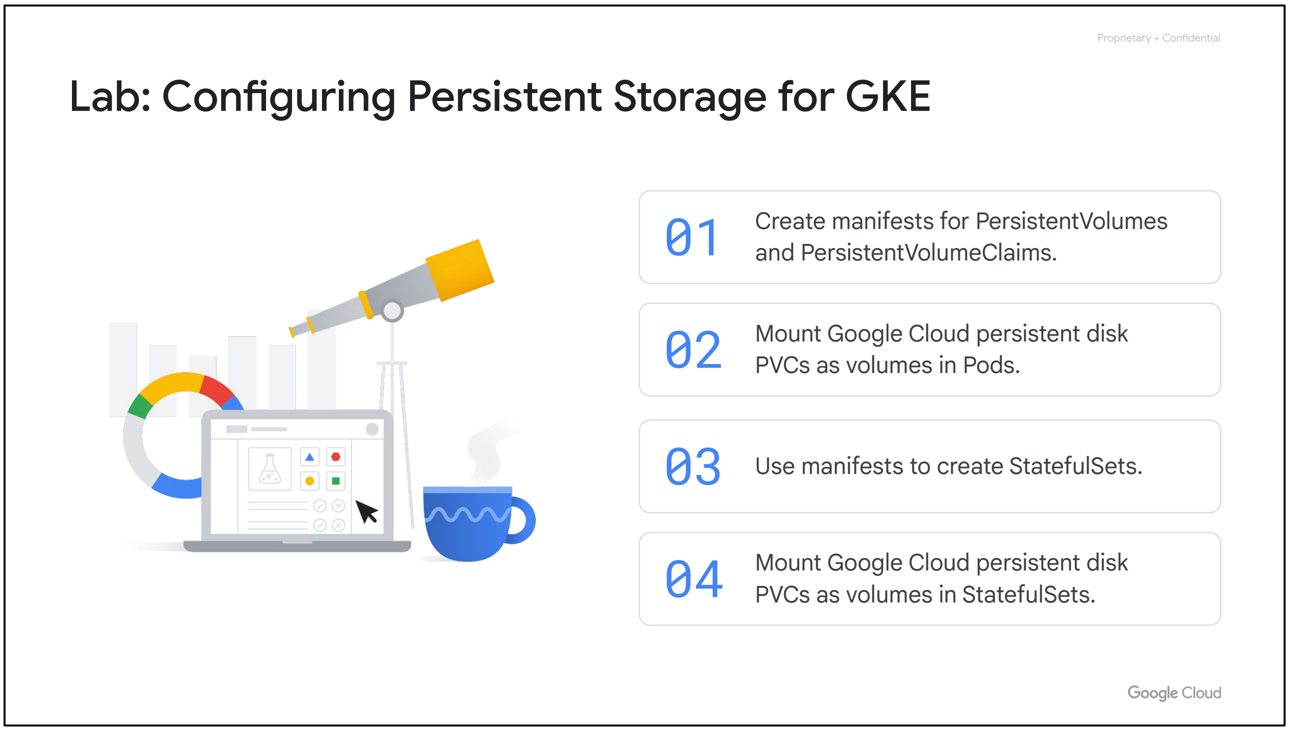
B. A TLS Secret

C. A JSON credential file

D. A Docker-Registry Secret ✓

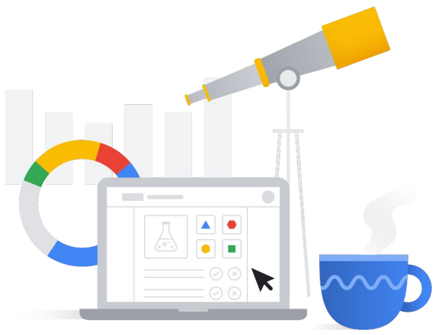
You need to store image registry credentials to allow Pods to pull images from a private repository. What type of Kubernetes Secret should you create?

A. A generic Secret: This is the **incorrect answer** because **generic secrets** are used for general purpose data and do not have a specific structure for storing image registry credentials..
B. A TLS Secret: This is the **incorrect answer** because **TLS secrets** are specifically designed for storing TLS certificates and keys, not image registry credentials..
C. A JSON credential file: This is the **incorrect answer** because **JSON file** does not match Kubernetes secrets' specific format for storing this type of sensitive data..
D. A Docker-Registry Secret: This is the **correct answer** because **Docker-Registry Secrets** are specifically designed to store the credentials needed to authenticate with private Docker registries. .



Proprietary + Confidential

Lab: Configuring Persistent Storage for GKE



01 Create manifests for PersistentVolumes and PersistentVolumeClaims.

02 Mount Google Cloud persistent disk PVCs as volumes in Pods.

03 Use manifests to create StatefulSets.

04 Mount Google Cloud persistent disk PVCs as volumes in StatefulSets.

Google Cloud

It’s time for some hands-on practice with GKE.

In the lab titled “Configuring Persistent Storage for Google Kubernetes Engine” you’ll:

- Create manifests for PersistentVolumes (PVs) and PersistentVolumeClaims (PVCs) for Google Cloud persistent disks (dynamically created or existing).
- Mount Google Cloud persistent disk PVCs as volumes in Pods.
- Use manifests to create StatefulSets.
- And mount Google Cloud 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.