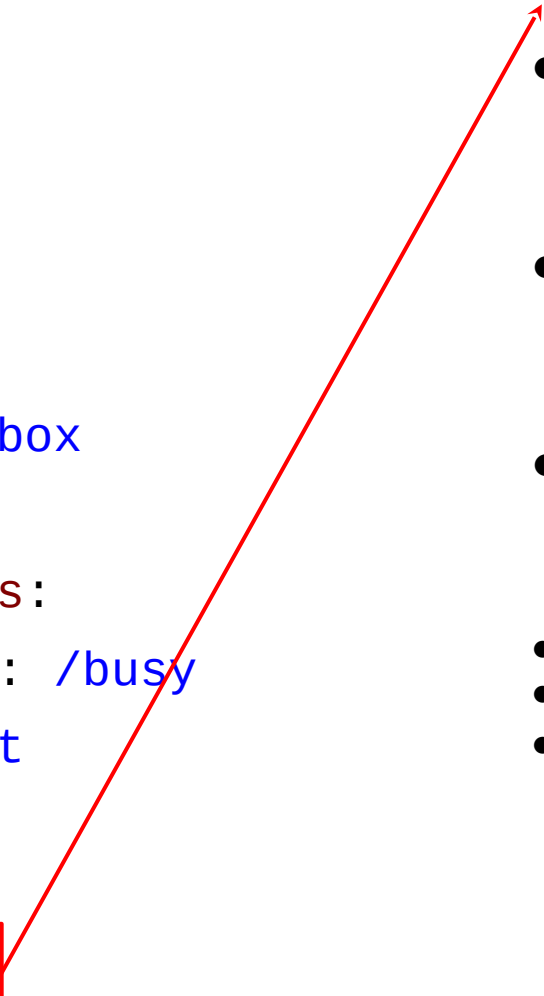


UNIDAD 5

ALMACENAMIENTO

Añadiendo persistencia a un Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: example
spec:
  containers:
  - image: busybox
    name: busy
    volumeMounts:
    - mountPath: /busy
      name: test
  volumes:
  - name: test
    <?????????>
```



Tenemos varios tipos de volúmenes:

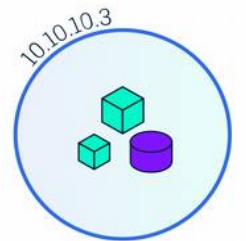
- **hostPath:** Este volumen corresponde a un directorio o fichero del nodo donde se crea el pod. No valido para cluster multinodo. Lo vamos a usar con minikube.
- **emptyDir:** El contenido de este volumen se borrará al eliminar el pod. Lo utilizamos para compartir información entre los contenedores de un mismo pod.
- **nfs:** Volumen en un sistema nfs (Network File System) compartido por todos los nodos del cluster.
- **gcePersistentDisk:** Google Compute Engine storage
- **awsElasticBlockStore:** AWS EBS storage
- ...

[Tipos de volúmenes que podemos utilizar](#)

Ejemplo

```
...  
containers:  
- name: nginx  
  image: nginx  
  volumeMounts:  
  - mountPath: /home  
    name: home  
  - mountPath: /nfs  
    name: nfs  
    readOnly: true  
  - mountPath: /temp  
    name: temp  
volumes:  
- name: home  
  hostPath:  
    path: /home/debian  
- name: nfs  
  nfs:  
    path: /  
    server: 10.0.0.2  
- name: temp  
  emptyDir: {}
```

- Se indica en el directorio del contenedor donde se monta cada volumen.
- Cada volumen es de un tipo
- El tipo emptyDir lo utilizamos para compartir información entre los contenedores de un mismo pod.



Desde el punto de vista del desarrollador



```
apiVersion: v1
kind: Pod
metadata:
  name: example
spec:
  containers:
  - image: busybox
    name: busy
    volumeMounts:
    - mountPath: /busy
      name: test
  volumes:
  - name: test
```

Volumen con 10Gb
Lectura y escritura

Un desarrollador **no debería conocer los distintos tipos de volúmenes disponibles en el cluster**. Son detalles muy específicos!!!

```
awsElasticBlockStore:
  volumeID: vol-0a07f3e37b
  fsType: ext4
```

```
hostPath:
  path: /data
```

```
gcePersistentDisk:
  pdName: my-data-disk
  fsType: ext4
```

- Un desarrollador se centra en indicar los requerimientos que debe tener el volumen que necesita:
 - Tamaño
 - Tipo de acceso (sólo lectura o lectura / escritura)
 - Tipo de volumen (sólo si es importante)
 - ...

¿Puede el cluster proporcionar un volumen con estas características?

PersistentVolumeClaims

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

El desarrollador hace una solicitud de almacenamiento usando un objeto del tipo *PersistentVolumenCliams*, indicando el **modo de acceso**:

- ReadWriteOnce
 - ReadOnlyMany
 - ReadWriteMany
- y el **tamaño** que necesita.

```
apiVersion: v1
kind: Pod
metadata:
  name: example
spec:
  containers:
    - image: busybox
      name: busy
      volumeMounts:
        - mountPath: /busy
          name: test
  volumes:
    - name: test
      persistentVolumeClaim:
        claimName: myclaim
```



PersistentVolume

- El **desarrollador** hace una **solicitud de almacenamiento**, indicando las características del volumen que necesita.
- Pero es el **administrador** será el responsable de dar de alta en el cluster los distintos volúmenes que hay disponibles, y que se representa con un recurso llamado **PersistentVolumen**.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0001
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  gcePersistentDisk:
    fsType: ext4
    pdName: pd-disk-1
```

Un *PersistentVolumen* es un objeto que representa los volúmenes disponibles en el cluster. En él se van a definir los detalles del backend de almacenamiento que vamos a utilizar, el tamaño disponible, los modos de acceso, las políticas de reciclaje, etc.

Tenemos tres modos de acceso, que depende del backend que vamos a utilizar:

- ReadWriteOnce: read-write solo para un nodo (RWO)
- ReadOnlyMany: read-only para muchos nodos (ROX)
- ReadWriteMany: read-write para muchos nodos (RWX)

Las políticas de reciclaje de volúmenes también depende del backend y son:

- Retain: El PV no se elimina, aunque el PVC se elimine. El administrador debe borrar el contenido para la próxima asociación.
- Recycle: Reutilizar contenido. Se elimina el contenido y el volumen es de nuevo utilizable.
- Delete: Se borra después de su utilización.

Backend de almacenamiento



Volume Plug-in	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
AWS EBS	✓	-	-
Azure File	✓	✓	✓
Azure Disk	✓	-	-
Ceph RBD	✓	✓	-
Fibre Channel	✓	✓	-
GCE Persistent Disk	✓	-	-
GlusterFS	✓	✓	✓
HostPath	✓	-	-
iSCSI	✓	✓	-
NFS	✓	✓	✓
Openstack Cinder	✓	-	-
VMWare vSphere	✓	-	-
Local	✓	-	-

Flujo de volúmenes. Método estático

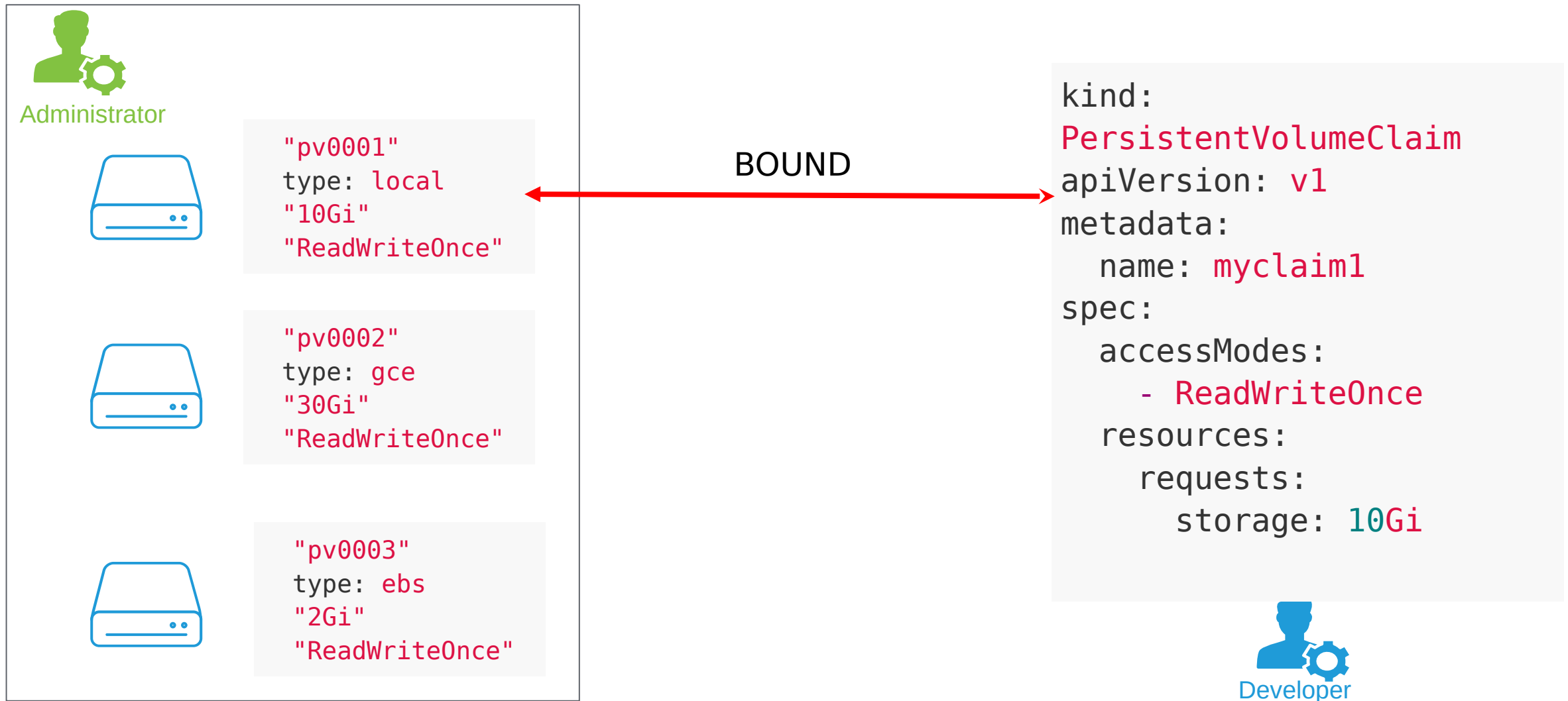


```
apiVersion: "v1"
kind:
  "PersistentVolume"
metadata:
  name: "pv0001"
  labels:
    type: local
spec:
  capacity:
    storage: "10Gi"
  accessModes:
    - "ReadWriteOnce"
  hostpath:
    path: "/mnt/data"
```

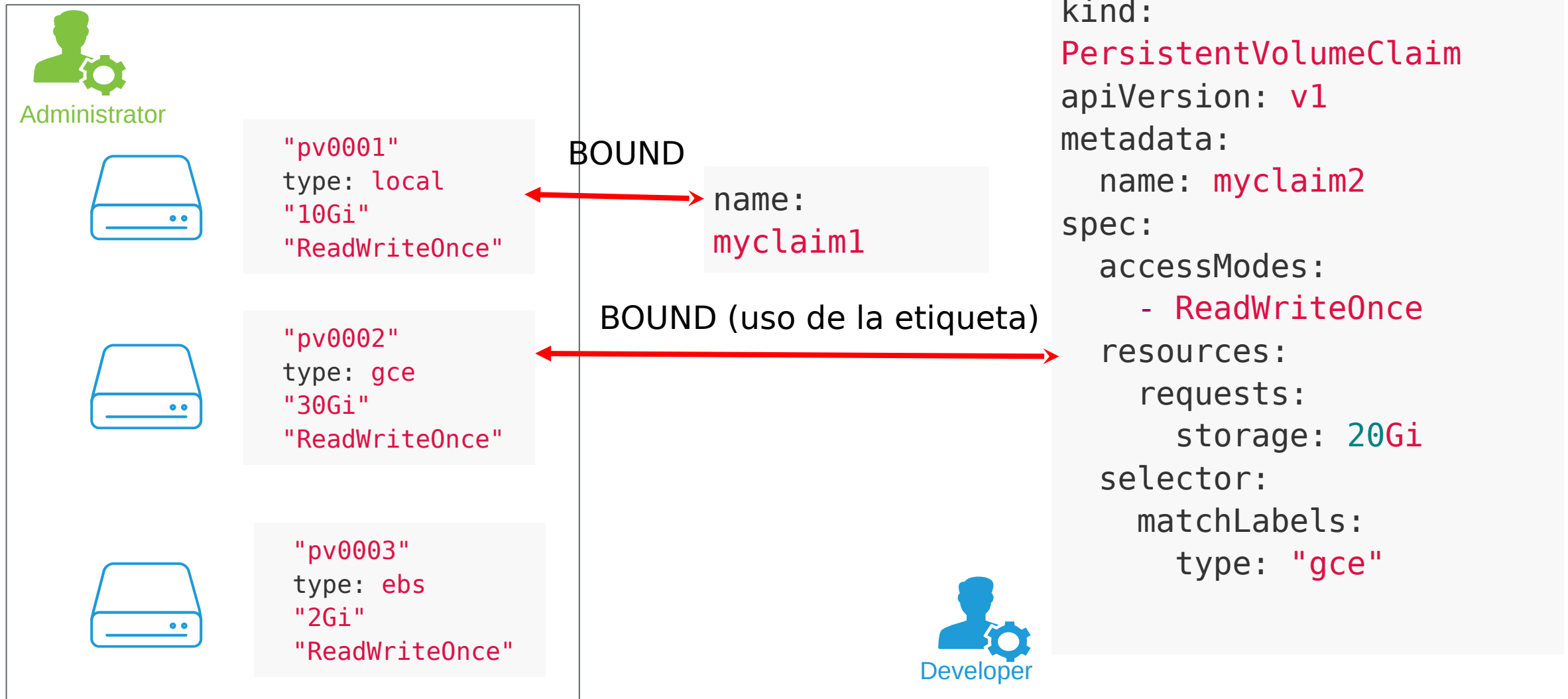
```
apiVersion: "v1"
kind:
  "PersistentVolume"
metadata:
  name: "pv0002"
  labels:
    type: gce
spec:
  capacity:
    storage: "30Gi"
  accessModes:
    - "ReadWriteOnce"
  gcePersistentDisk:
    fsType: "ext4"
    pdName: "pd-disk-
1"
```

```
apiVersion: "v1"
kind: "PersistentVolume"
metadata:
  name: "pv0003"
  labels:
    type: ebs
spec:
  capacity:
    storage: "2Gi"
  accessModes:
    - "ReadWriteOnce"
  awsElasticBlockStore:
    volumeID: vol-
0a07f3e37b
    fsType: ext4
```

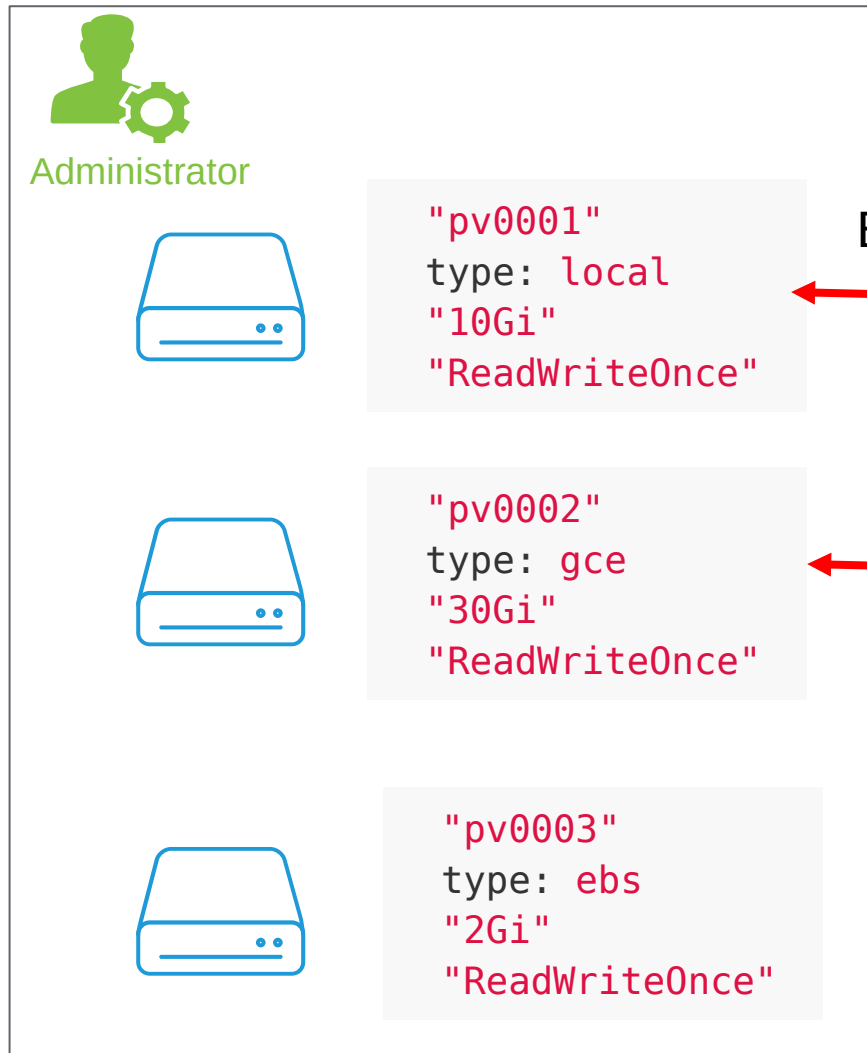

Flujo de volúmenes. Método estático



Flujo de volúmenes. Método estático



Flujo de volúmenes. Método estático



BOUND

name:
myclaim1

BOUND

name:
myclaim2

No volume
available

```
kind:  
PersistentVolumeClaim  
apiVersion: v1  
metadata:  
  name: myclaim3  
spec:  
  accessModes:  
    - ReadWriteOnce  
  resources:  
    requests:  
      storage: 100Gi
```



Flujo de volúmenes. Método dinámico

Podemos crear volúmenes de forma dinámica a partir de una petición de volumen. En este caso necesitamos un “**provisionador**” de almacenamiento (para cada uno de los backend), de tal manera que cada vez que se cree un **PersistentVolumeClaim**, se creará bajo demanda un **PersistentVolume** que se ajuste a las características seleccionadas.

Los administradores pueden usar el objeto [StorageClass](#) (plugin de almacenamiento) para obtener la [gestión dinámica de volúmenes](#).

En minikube:

```
kubectl get storageclass
NAME                                PROVISIONER                AGE
standard (default)                 k8s.io/minikube-hostpath   2d23h
```

Tenemos un provisionador de almacenamiento local del tipo hostPath.

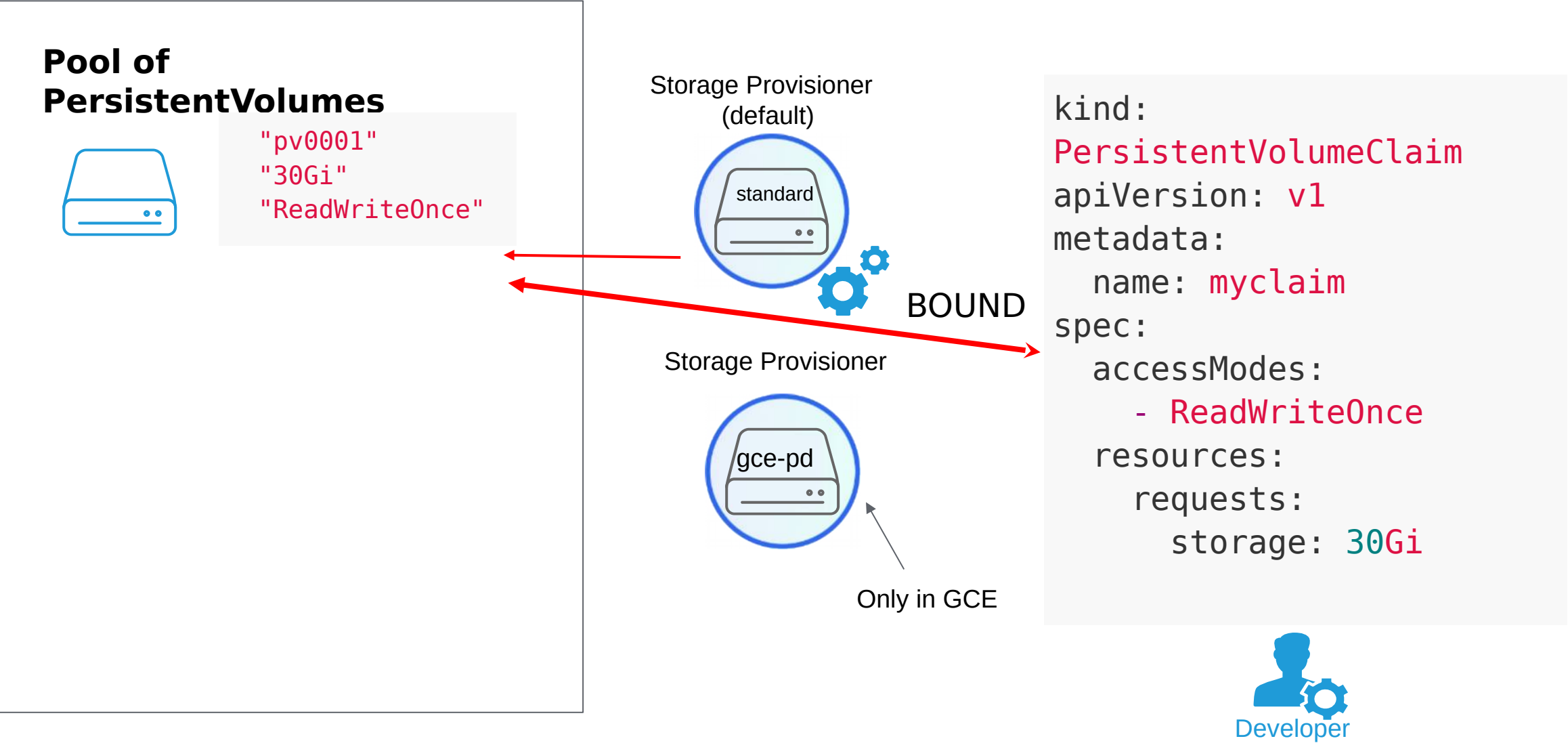
```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc1
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

Gestión Estática

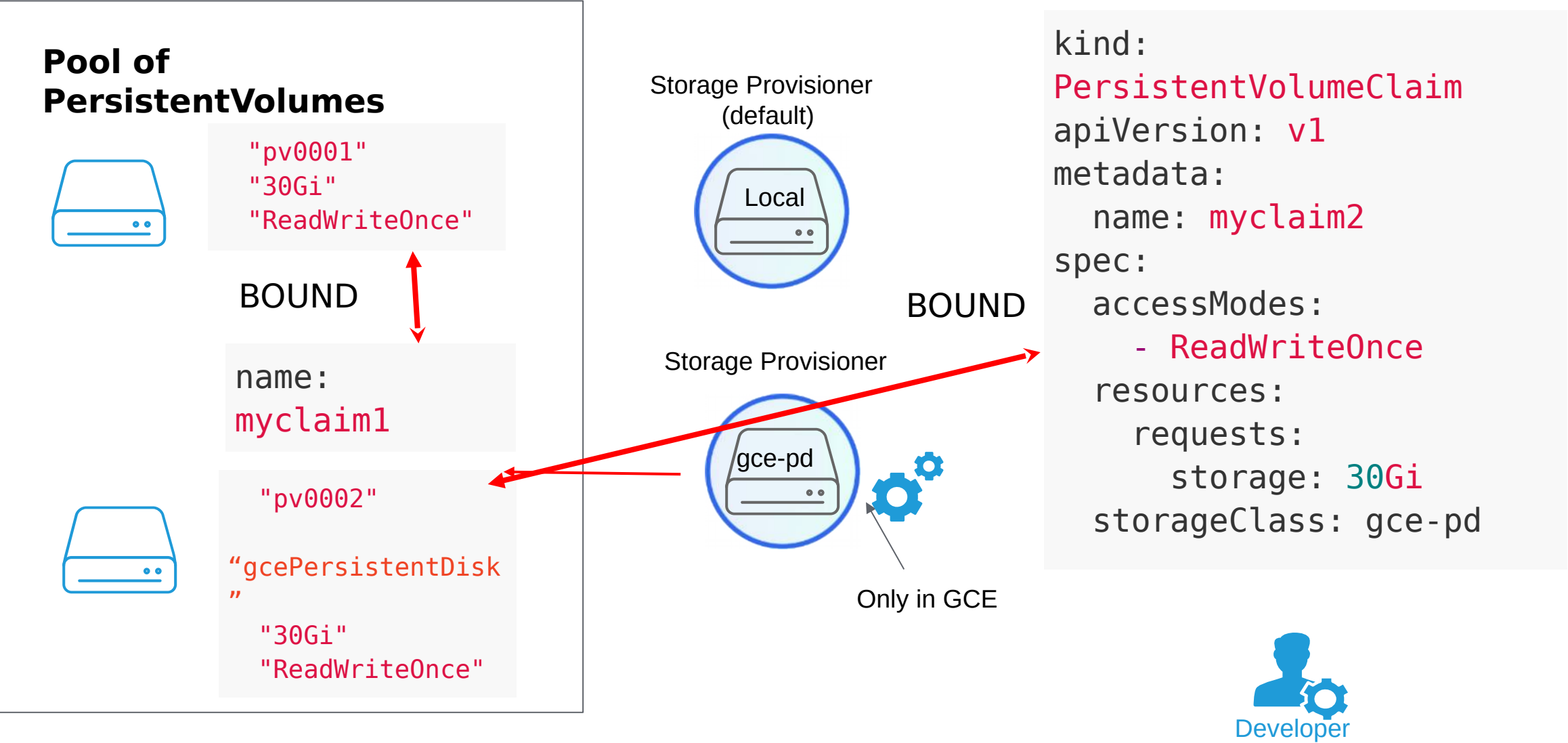
Gestión Dinámica
(No se indica el StorageClass)

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc2
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

Flujo de volúmenes. Método dinámico



Flujo de volúmenes. Método dinámico



EJEMPLO 1. Creación de PV y PVC (estático)

- Creamos el PersistentVolume:

```
kubectl create -f pv.yaml
```

```
kubectl get pv
```

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM	STORAGECLASS	REASON	AGE
pv1	5Gi	RWX	Recycle	Available		manual		5s

- Creamos el PersistentVolumeClaim:

```
kubectl create -f pvc1.yaml
```

```
kubectl get pv
```

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM	STORAGECLASS	REASON	AGE
pv1	5Gi	RWX	Recycle	Bound	default/pvc1	manual		66s

```
kubectl get pvc
```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
pvc1	Bound	pv1	5Gi	RWX	manual	31s

EJEMPLO 1. Creación de PV y PVC (estático)

- Escribimos un fichero index.html en el directorio correspondiente al volumen:

```
minikube ssh  
$ sudo sh -c "echo 'Hello from Kubernetes storage' > /data/pv1/index.html"
```

- Creamos un pod con el volumen

```
kubectl create -f pod.yaml  
kubectl get pod  
kubectl describe pod task-pv-pod
```

- Accedemos al pod, instalamos curl y probamos a acceder al servidor web

```
kubectl exec -it task-pv-pod -- /bin/bash
```

```
root@task-pv-pod:/# apt-get update  
root@task-pv-pod:/# apt-get install curl  
root@task-pv-pod:/# curl localhost  
Hello from Kubernetes storage
```


EJEMPLO 2. Gestión dinámica de volúmenes

En minikube, tenemos un provisionador de almacenamiento local del tipo hostPath.

```
kubectl get storageclass
NAME                                PROVISIONER                AGE
standard (default)                 k8s.io/minikube-hostpath  2d23h
```

Por la tanto si creamos un PersistentVolumeClaim, se creará de forma dinámica un PV:

```
kubectl create -f pvc2.yaml
```

```
kubectl get pv
NAME                                CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS  CLAIM              STORAGECLASS
pvc-fba7b6e9-817e-11e9-9dd3-080027b9a41d  1Gi      RWX           Delete          Bound   default/pvc2       standard
```

```
kubectl get pvc
NAME    STATUS  VOLUME                                CAPACITY  ACCESS MODES  STORAGECLASS  AGE
pvc2    Bound  pvc-fba7b6e9-817e-11e9-9dd3-080027b9a41d  1Gi      RWX           standard      15s
```

```
kubectl delete pvc pvc2
kubectl get pv
```

EJEMPLO 3. WordPress con almacenamiento persistente

```
kubectl create -f wordpress-pv.yaml
kubectl create -f wordpress-ns.yaml

kubectl create -f wordpress-pvc.yaml
kubectl create -f mariadb-pvc.yaml
kubectl get pv,pvc -n wordpress

kubectl apply -f mariadb-srv.yaml
kubectl apply -f wordpress-srv.yaml
kubectl apply -f wordpress-ingress.yaml
kubectl apply -f mariadb-secret.yaml
kubectl apply -f mariadb-deployment.yaml
kubectl apply -f wordpress-deployment.yaml

kubectl get all -n wordpress
```

