What are Data & Database?

Suppose a company needs to store the names of hundreds of employees working in the company in such a way that all the employees can be individually identified. Then, the company collects the data of all those employees. Now, when I say data, I mean that the company collects distinct pieces of information about an object. So, that object could be a real-world entity such as people, or any object such as a mouse, laptop etc.

Now, when you have such a large amount of data, you obviously need a place to store it, which is a Database.

Database Management System & Types of DBMS

A Database Management System (DBMS) is a software application that interacts with the user, applications and the database itself to capture and analyze data. The data stored in the database can be modified, retrieved and deleted, and can be of any type like strings, numbers, images etc.

Types of DBMS

There are mainly 4 types of DBMS, which are Hierarchical, Relational, Network, and Object-Oriented DBMS.

- Hierarchical DBMS: As the name suggests, this type of DBMS has a style of predecessor-successor type of relationship. So, it has a structure similar to that of a tree, wherein the nodes represent records and the branches of the tree represent fields.
- Relational DBMS (RDBMS): This type of DBMS, uses a structure that allows the users to identify and access data *in relation* to another piece of data in the database.
- Network DBMS: This type of DBMS supports many to many relations wherein multiple member records can be linked.
- Object-oriented DBMS: This type of DBMS uses small individual software called objects. Each object contains a piece of data, and the instructions for the actions to be done with the data.

Structured Query Language (SQL)

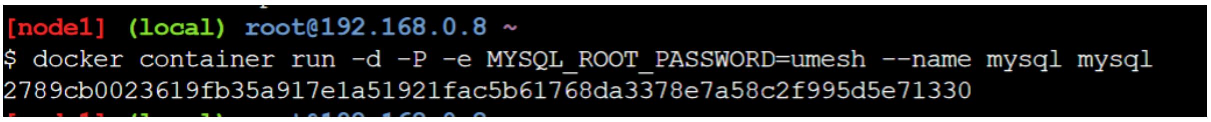
SQL is the core of a relational database which is used for accessing and managing the database. By using SQL, you can add, update or delete rows of data, retrieve subsets of information, modify databases and perform many actions. The different subsets of SQL are as follows:

- *DDL (Data Definition Language)* – It allows you to perform various operations on the database such as CREATE, ALTER and DELETE objects.
- *DML (Data Manipulation Language)* – It allows you to access and manipulate data. It helps you to insert, update, delete and retrieve data from the database.
- *DCL (Data Control Language)* – It allows you to control access to the database. Example – Grant or Revoke access permissions.
- *TCL (Transaction Control Language)* – It allows you to deal with the transaction of the database. Example – Commit, Rollback, Savepoint, Set Transaction.

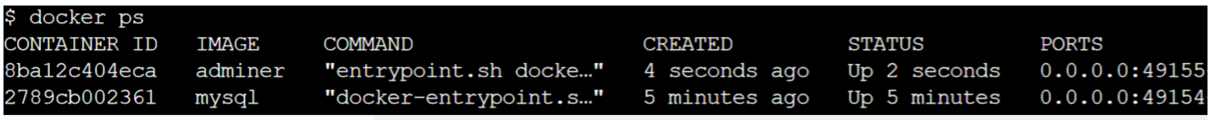
MySQL Docker support page: https://hub.docker.com/_/mysql

docker container run -d -P -e MYSQL_ROOT_PASSWORD=umesh --name mysql mysql

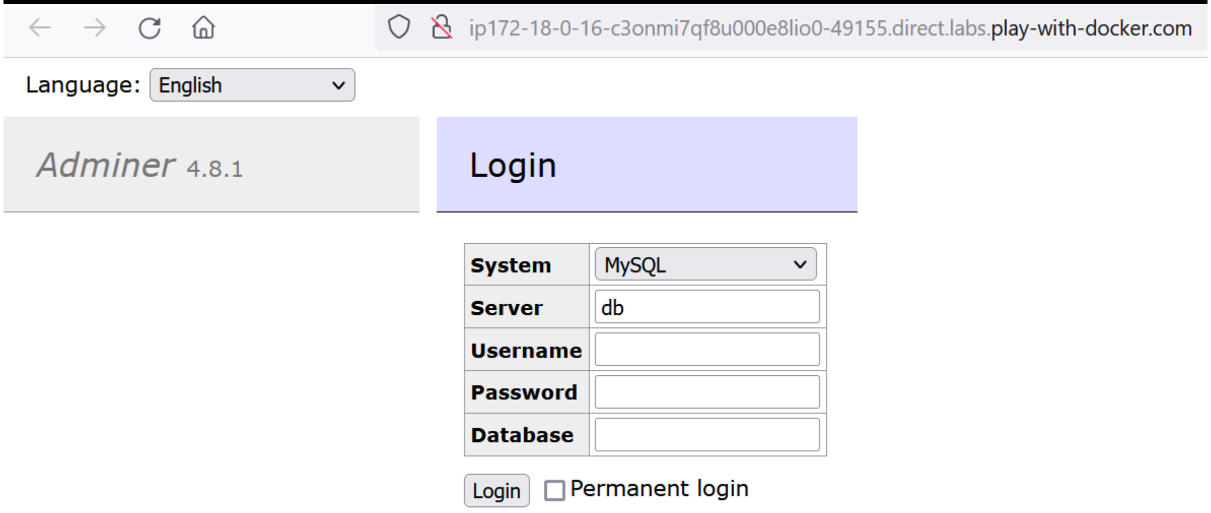
docker container run -d -P --name adminer adminer



```
[node1] (local) root@192.168.0.8 ~
$ docker container run -d -P -e MYSQL_ROOT_PASSWORD=umesh --name mysql mysql
2789cb0023619fb35a917e1a51921fac5b61768da3378e7a58c2f995d5e71330
```



```
$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS
8ba12c404eca   adminer   "entrypoint.sh docke…"   4 seconds ago   Up 2 seconds   0.0.0.0:49155
2789cb002361   mysql     "docker-entrypoint.s…"   5 minutes ago   Up 5 minutes   0.0.0.0:49154
```



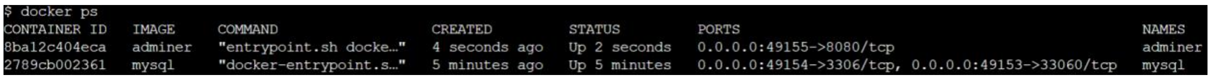
ip172-18-0-16-c3onmi7qf8u000e8lio0-49155.direct.labs.play-with-docker.com

Language: English

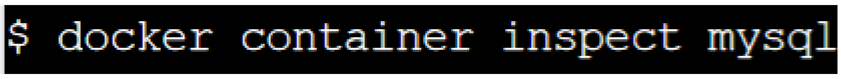
Adminer 4.8.1

Login

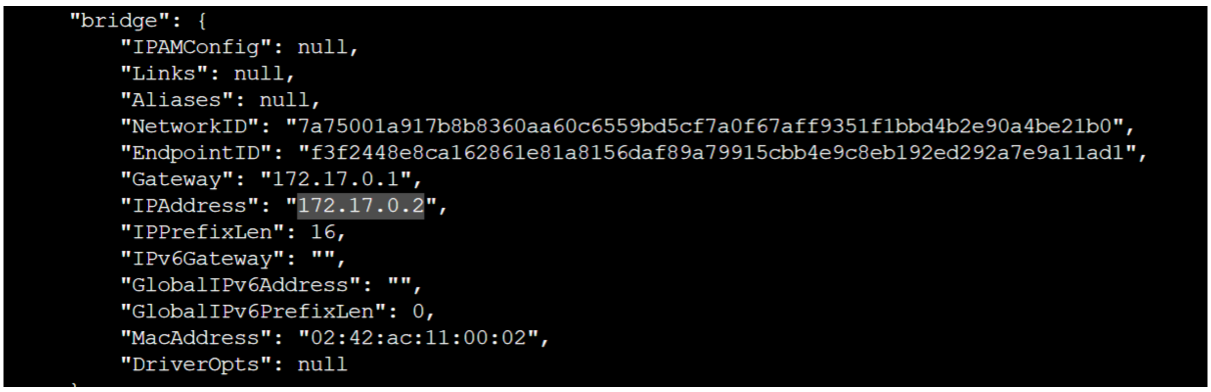
System	MySQL
Server	db
Username	
Password	
Database	



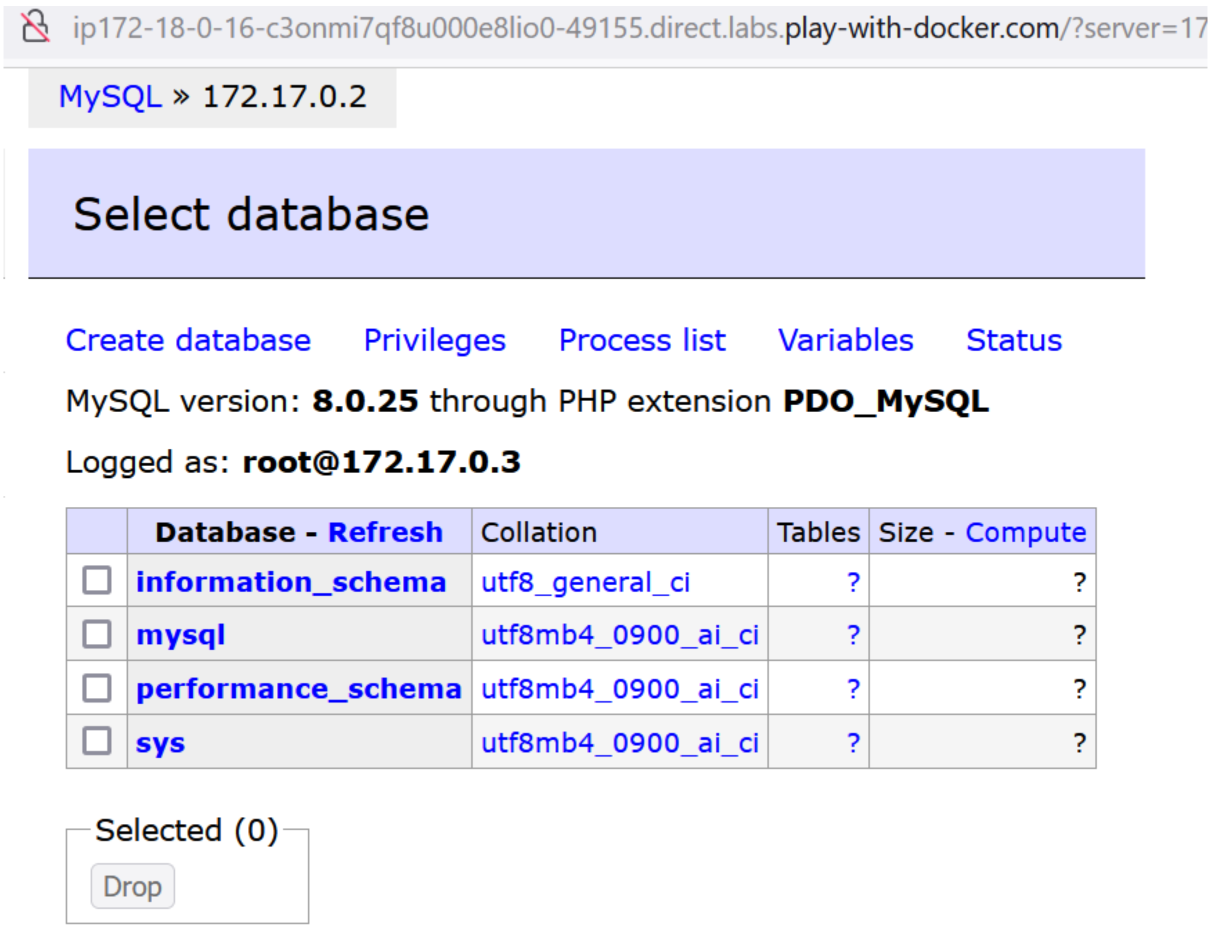
```
$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS                                               NAMES
8ba12c404eca   adminer   "entrypoint.sh docke…"   4 seconds ago   Up 2 seconds   0.0.0.0:49155->8080/tcp                             adminer
2789cb002361   mysql     "docker-entrypoint.s…"   5 minutes ago   Up 5 minutes   0.0.0.0:49154->3306/tcp, 0.0.0.0:49153->33060/tcp   mysql
```



```
$ docker container inspect mysql
```



```
    "bridge": {
        "IPAMConfig": null,
        "Links": null,
        "Aliases": null,
        "NetworkID": "7a75001a917b8b8360aa60c6559bd5cf7a0f67aff9351f1bbd4b2e90a4be21b0",
        "EndpointID": "f3f2448e8ca162861e81a8156daf89a79915cbb4e9c8eb192ed292a7e9a11ad1",
        "Gateway": "172.17.0.1",
        "IPAddress": "172.17.0.2",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "02:42:ac:11:00:02",
        "DriverOpts": null
```



ip172-18-0-16-c3onmi7qf8u000e8lio0-49155.direct.labs.play-with-docker.com/?server=17

MySQL » 172.17.0.2

Select database

Create database Privileges Process list Variables Status

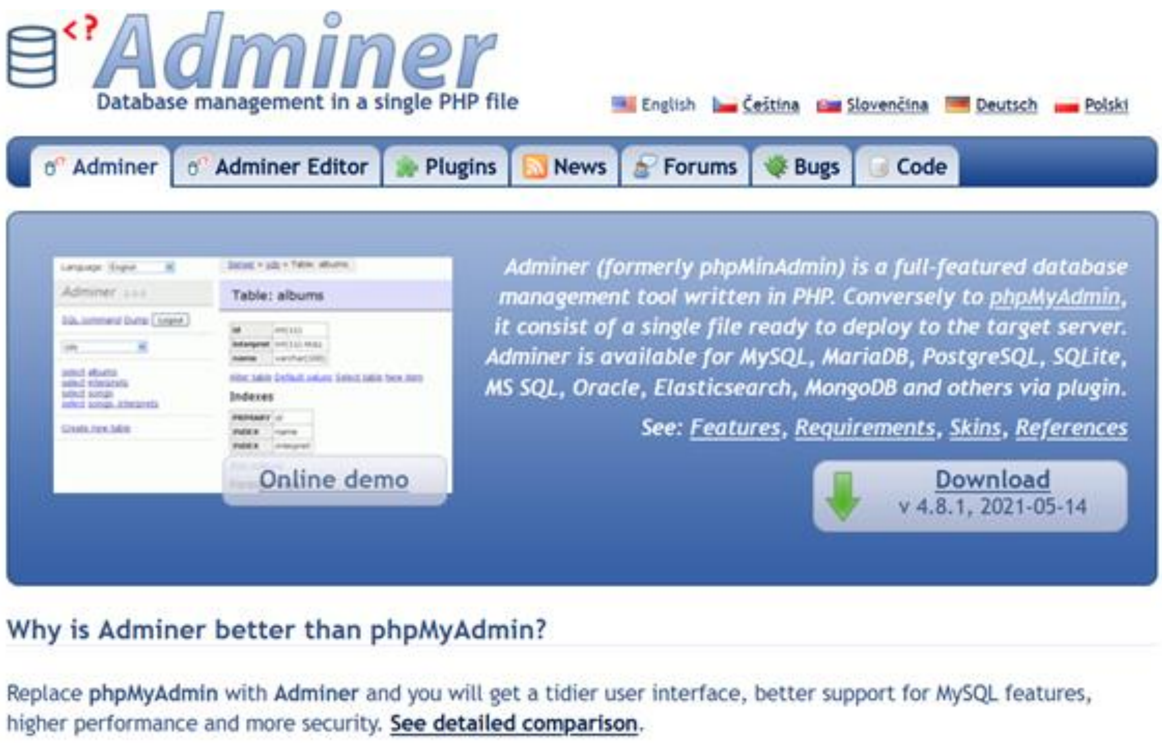
MySQL version: **8.0.25** through PHP extension **PDO_MySQL**

Logged as: **root@172.17.0.3**

	Database - Refresh	Collation	Tables	Size - Compute
☐	**information_schema**	utf8_general_ci	?	?
☐	**mysql**	utf8mb4_0900_ai_ci	?	?
☐	**performance_schema**	utf8mb4_0900_ai_ci	?	?
☐	**sys**	utf8mb4_0900_ai_ci	?	?

Selected (0)

Drop



Adminer (formerly phpMinAdmin) is a full-featured database management tool written in PHP. Conversely to phpMyAdmin, it consist of a single file ready to deploy to the target server. Adminer is available for MySQL, MariaDB, PostgreSQL, SQLite, MS SQL, Oracle, Elasticsearch, MongoDB and others via plugin.

See: Features, Requirements, Skins, References

Why is Adminer better than phpMyAdmin?

Replace **phpMyAdmin** with **Adminer** and you will get a tidier user interface, better support for MySQL features, higher performance and more security. **See detailed comparison**.

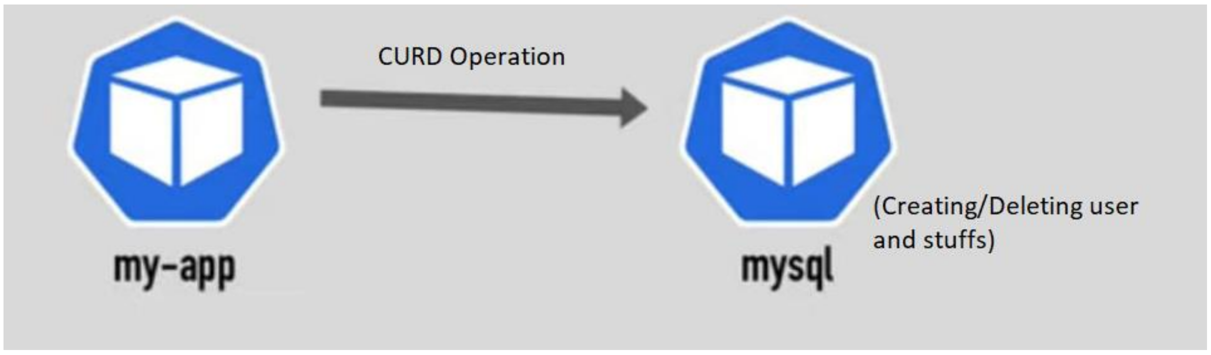
If mySQL pod goes down then we lost all data. For addressing this issue we have Kubernetes volume concept.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysqlclaim
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 1Gi
  storageClassName: standard
```

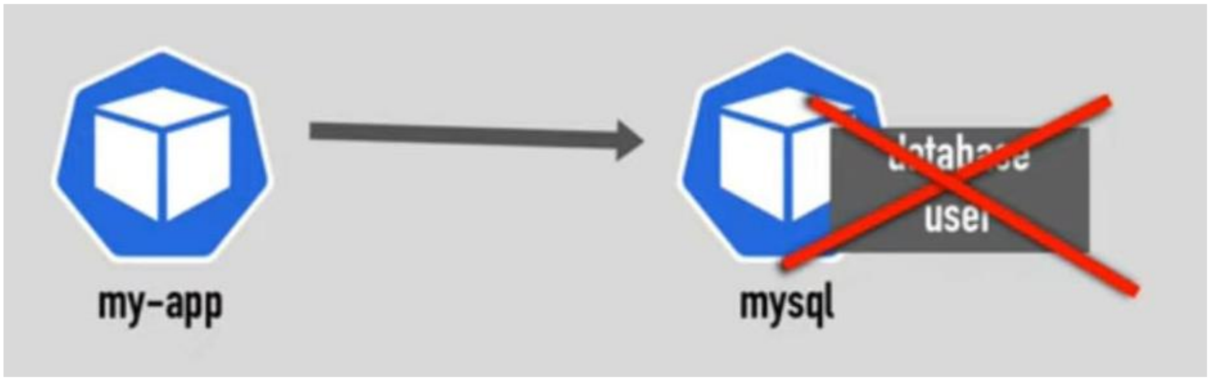
```yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: mysql
spec:
  containers:
    - image: mysql
      name: mysql
      env:
        - name: MYSQL_ROOT_PASSWORD
          value: qwert456
      ports:
        - containerPort: 3306
          name: mysql
      volumeMounts:
        - name: mysql-store
          mountPath: /var/lib/mysql
  volumes:
    - name: mysql-store
      persistentVolumeClaim:
        claimName: mysqlclaim
```

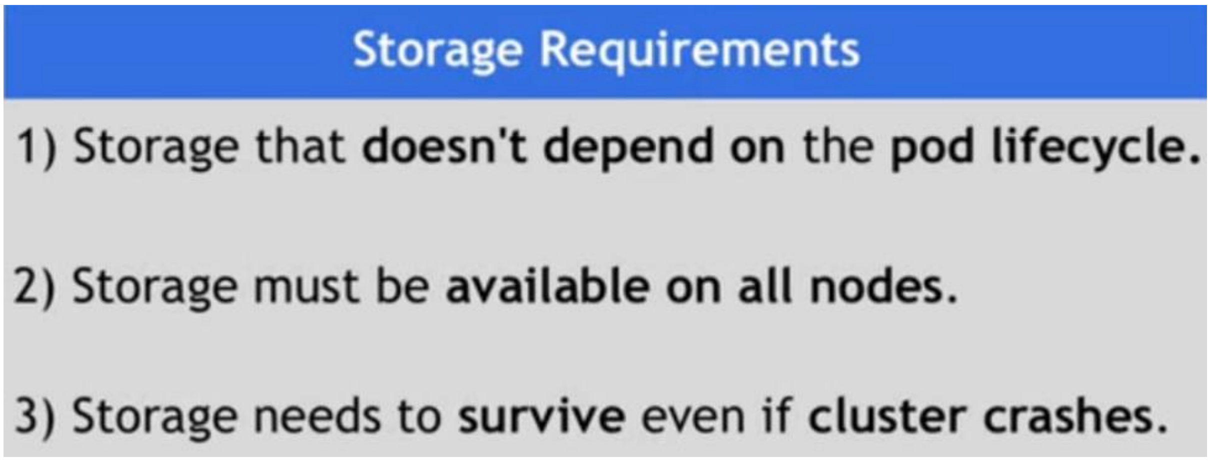
Persistent Volume
pv
Persistent Volume Claim
pvc
Storage Class
sc

Let's consider a scenario where we have microservice application running in K8s Environment. Application can perform CURD operation whereas there are chances that few Database admin activity would be done at DB side.



Let's say database pod is down due to some issue, and we lost everything.





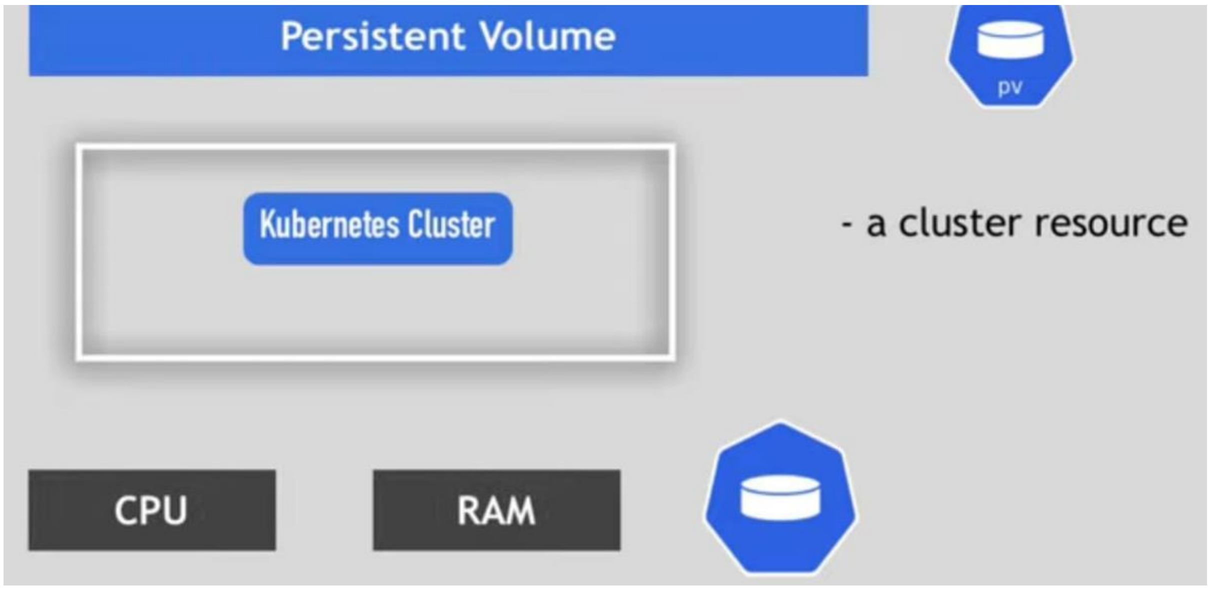
Storage Requirements

1) Storage that **doesn't depend on the pod lifecycle.**

2) Storage must be **available on all nodes.**

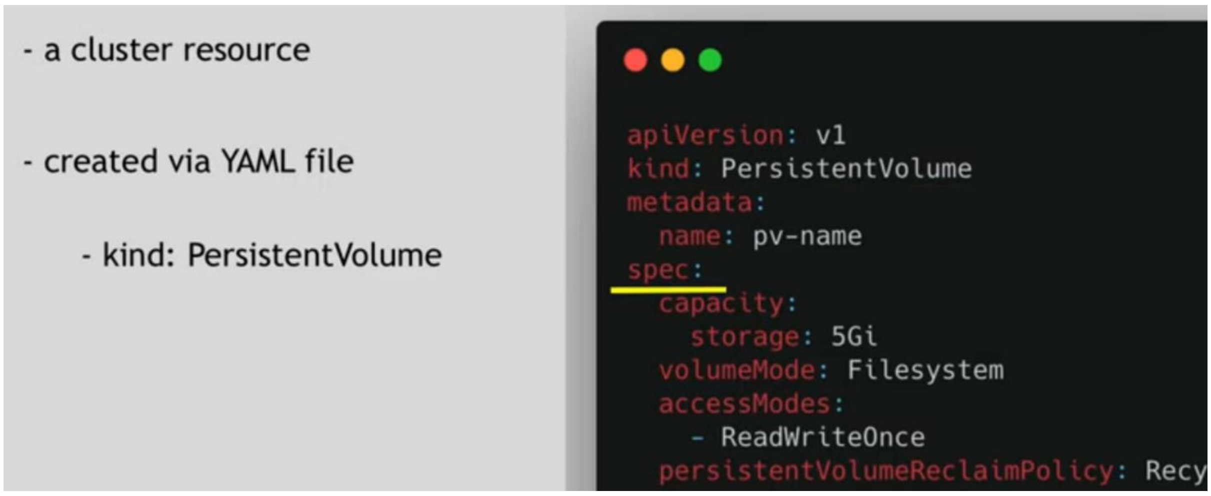
3) Storage needs to **survive** even if **cluster crashes.**

Note: We are not sure which pod may go down.



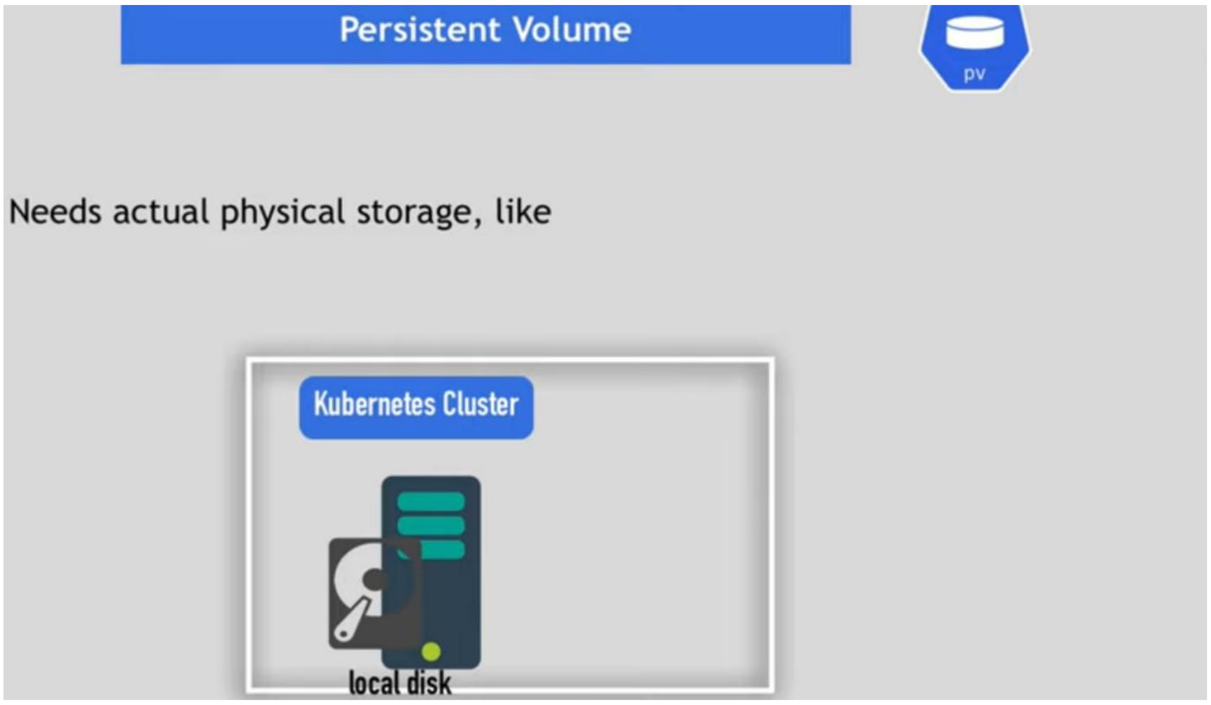
Persistent Volume

- a cluster resource



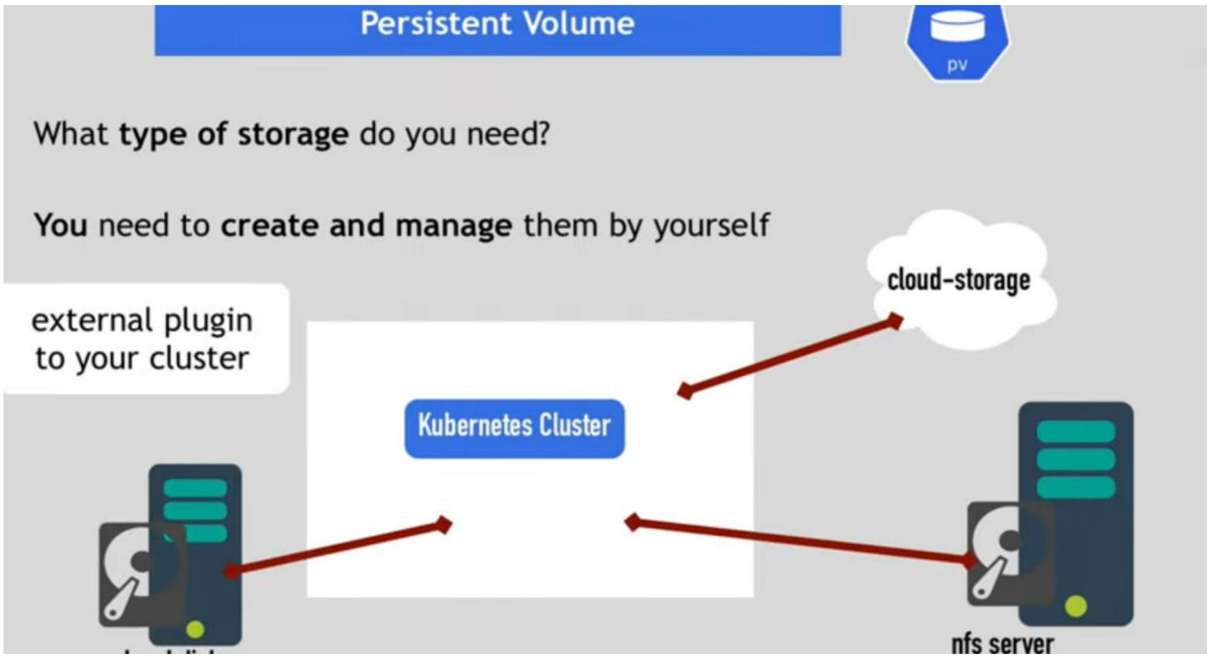
- a cluster resource
- created via YAML file
    - kind: PersistentVolume

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-name
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recy
```



Persistent Volume

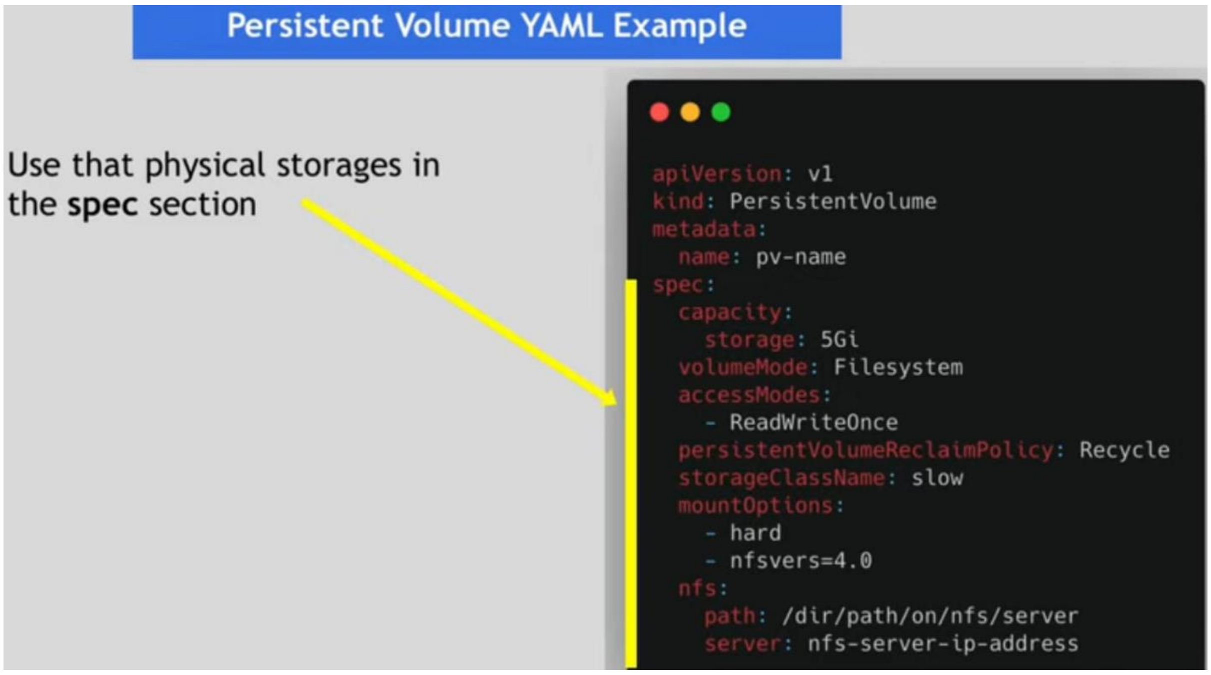
Needs actual physical storage, like



Persistent Volume

What **type of storage** do you need?

You need to **create and manage** them by yourself



Persistent Volume YAML Example

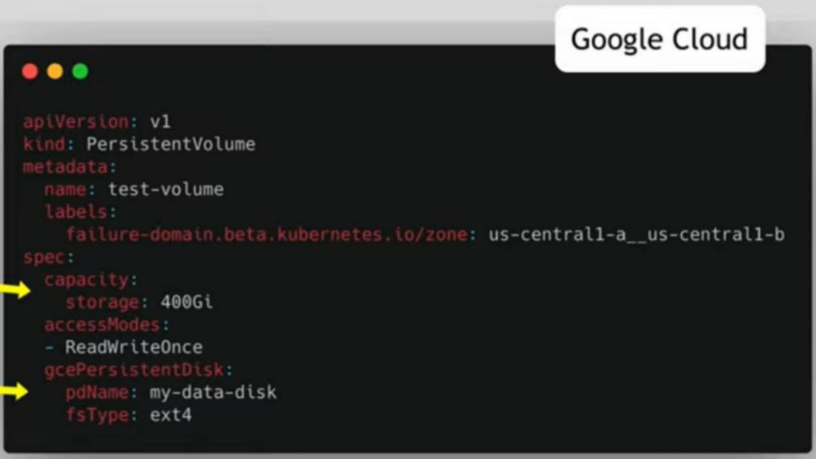
Use that physical storages in the **spec** section

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-name
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.0
  nfs:
    path: /dir/path/on/nfs/server
    server: nfs-server-ip-address
```

Persistent Volume YAML Example

How much:

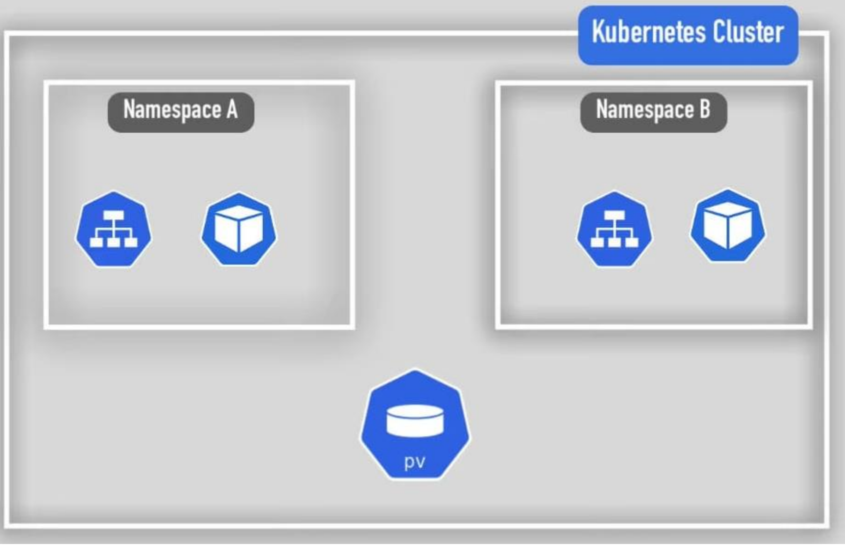
Google Cloud parameters:



```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: test-volume
  labels:
    failure-domain.beta.kubernetes.io/zone: us-central1-a__us-central1-b
spec:
  capacity:
    storage: 400Gi
  accessModes:
  - ReadWriteOnce
  gcePersistentDisk:
    pdName: my-data-disk
    fsType: ext4
```

Persistent Volumes are **NOT namespaced**

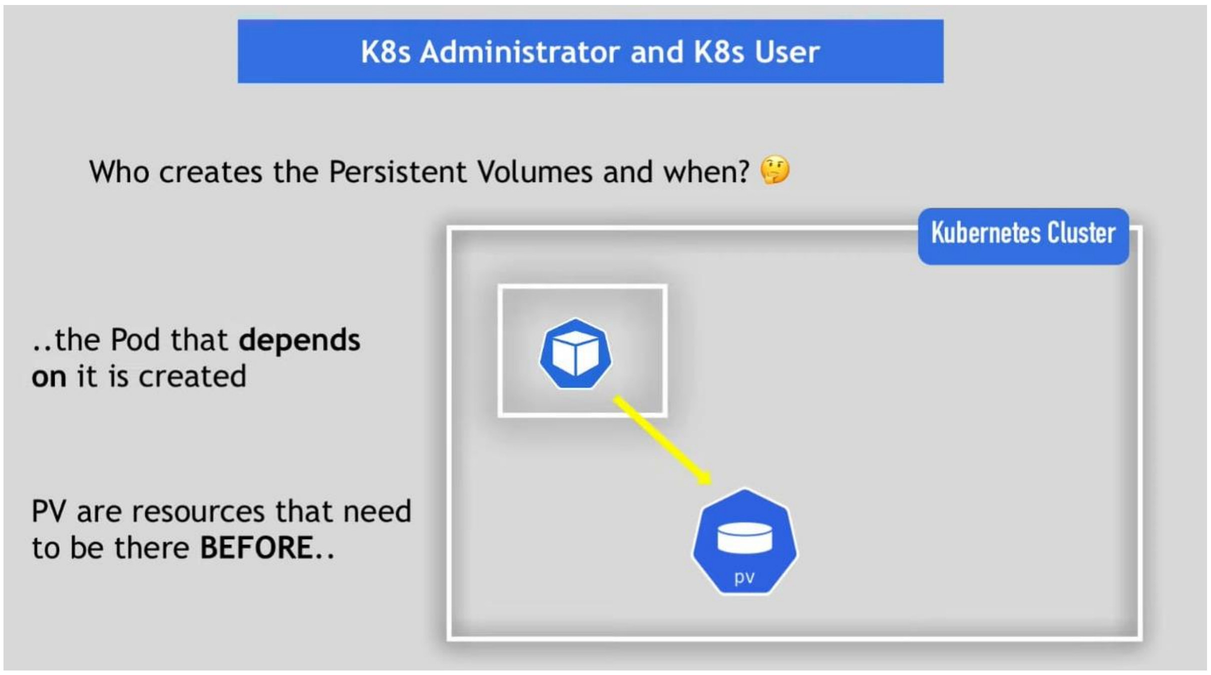
PV outside of the namespaces



Local vs. Remote Volume Types

Each volume type has it's own use case!

Local volume types violate 2. and 3. requirement for data persistence:

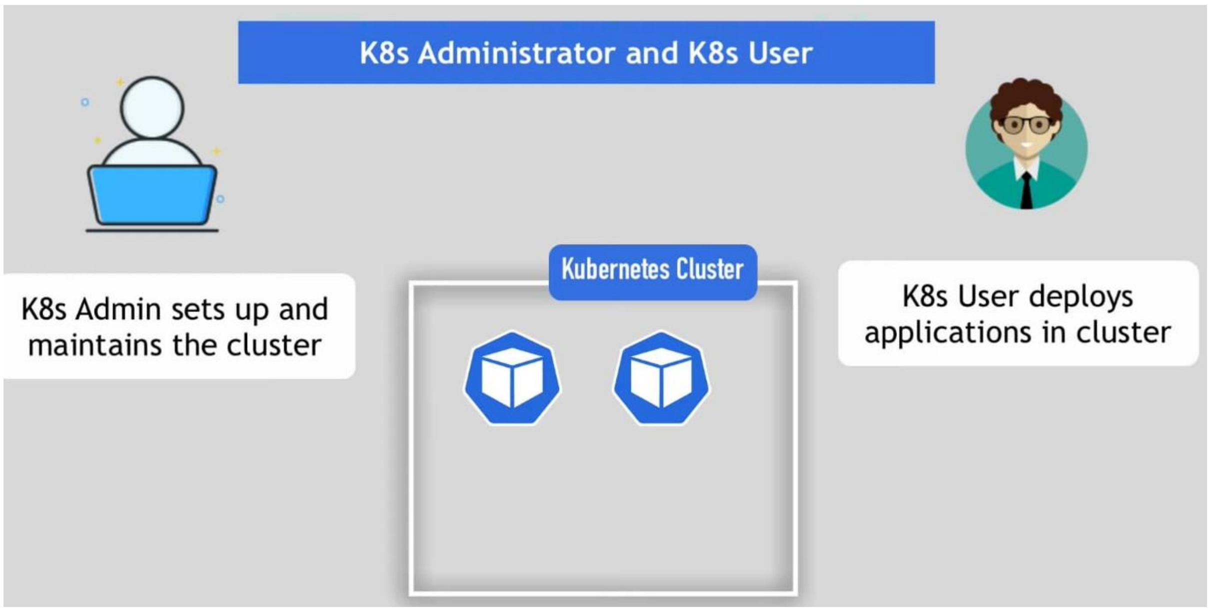


K8s Administrator and K8s User

Who creates the Persistent Volumes and when? 🤔

..the Pod that **depends on** it is created

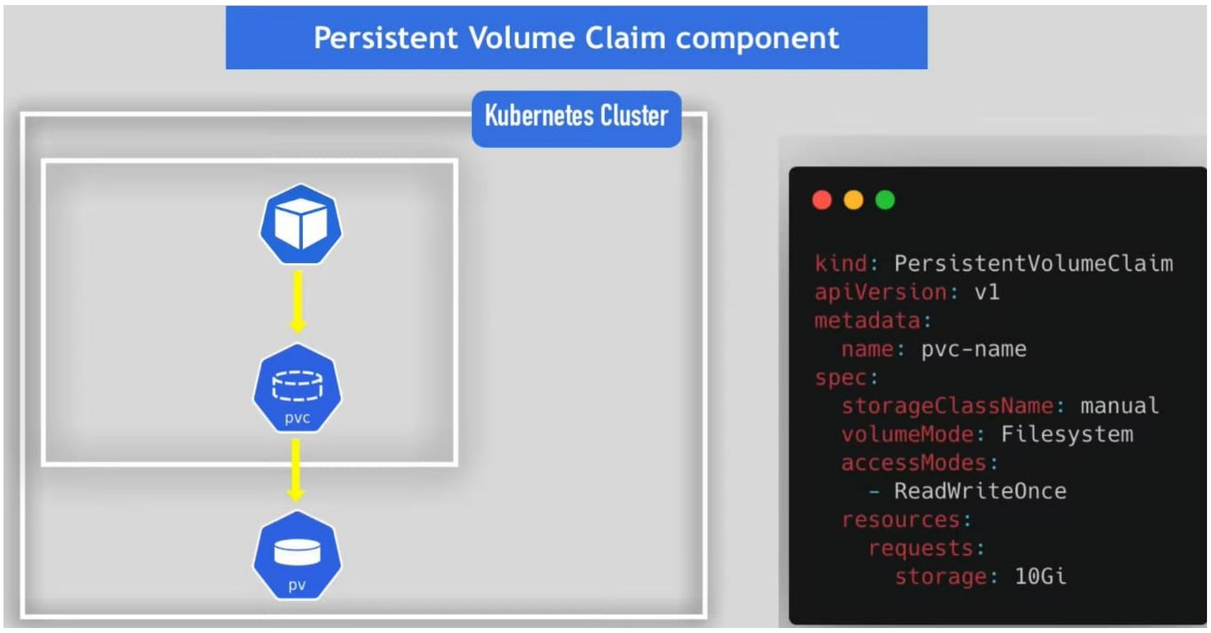
PV are resources that need to be there **BEFORE**..



K8s Administrator and K8s User

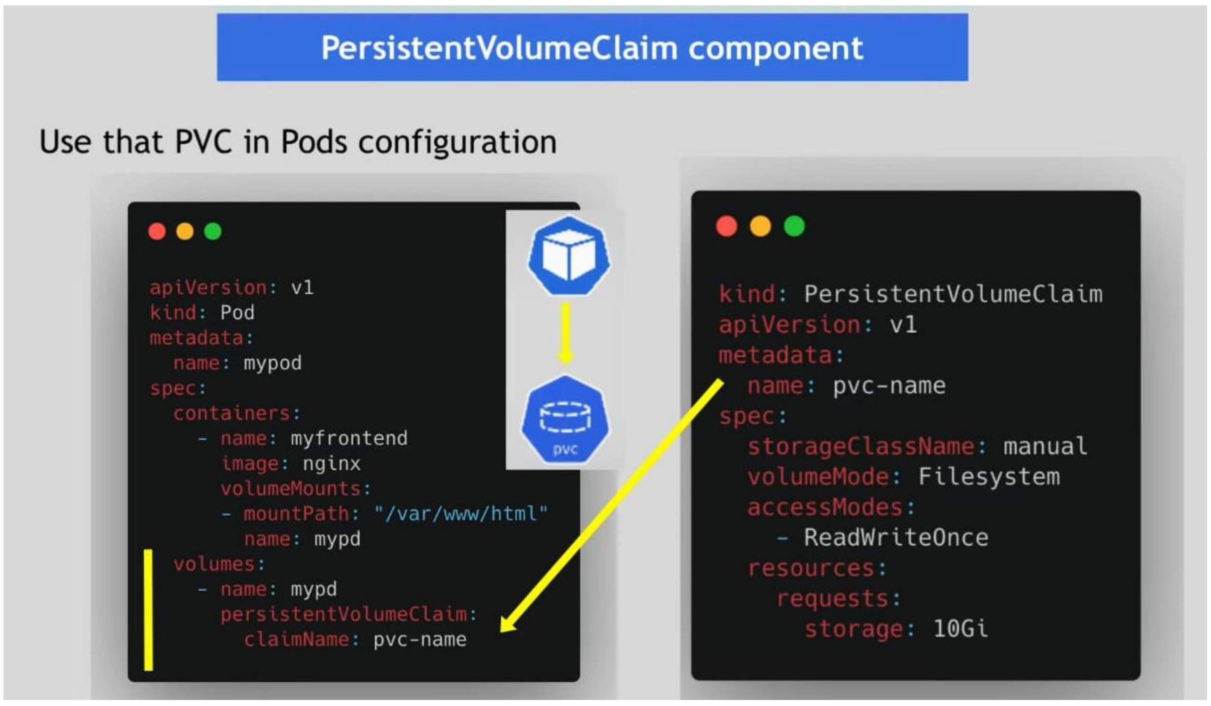
K8s Admin sets up and maintains the cluster

K8s User deploys applications in cluster



Persistent Volume Claim component

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc-name
spec:
  storageClassName: manual
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

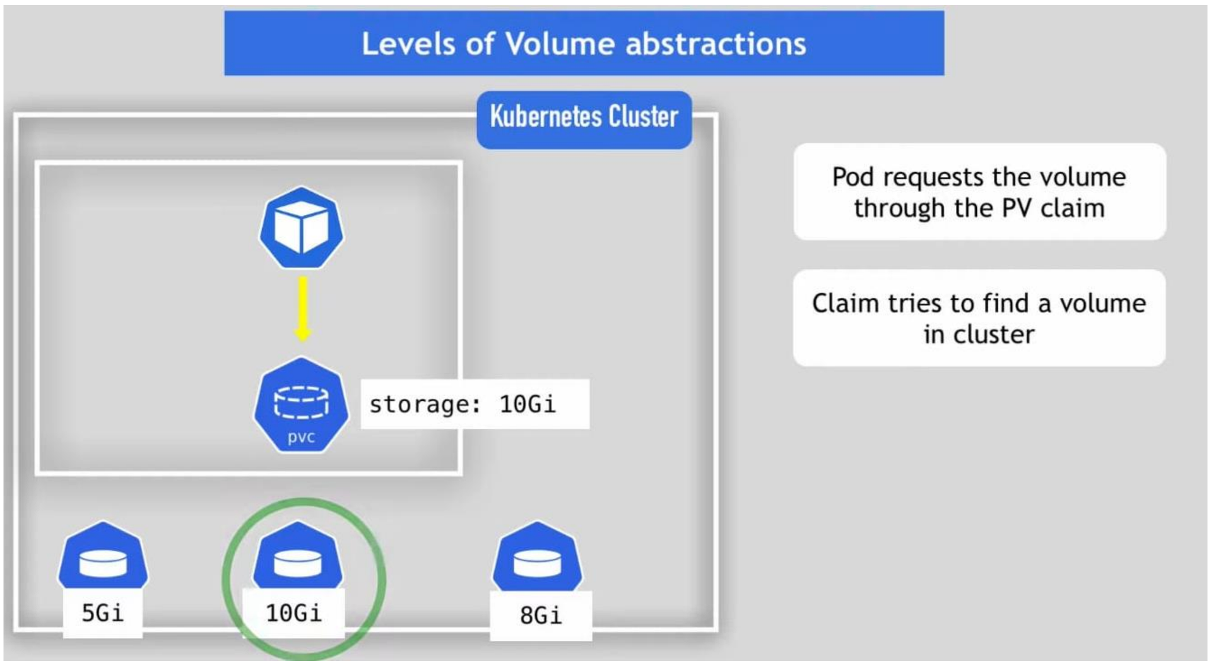


PersistentVolumeClaim component

Use that PVC in Pods configuration

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: pvc-name
```

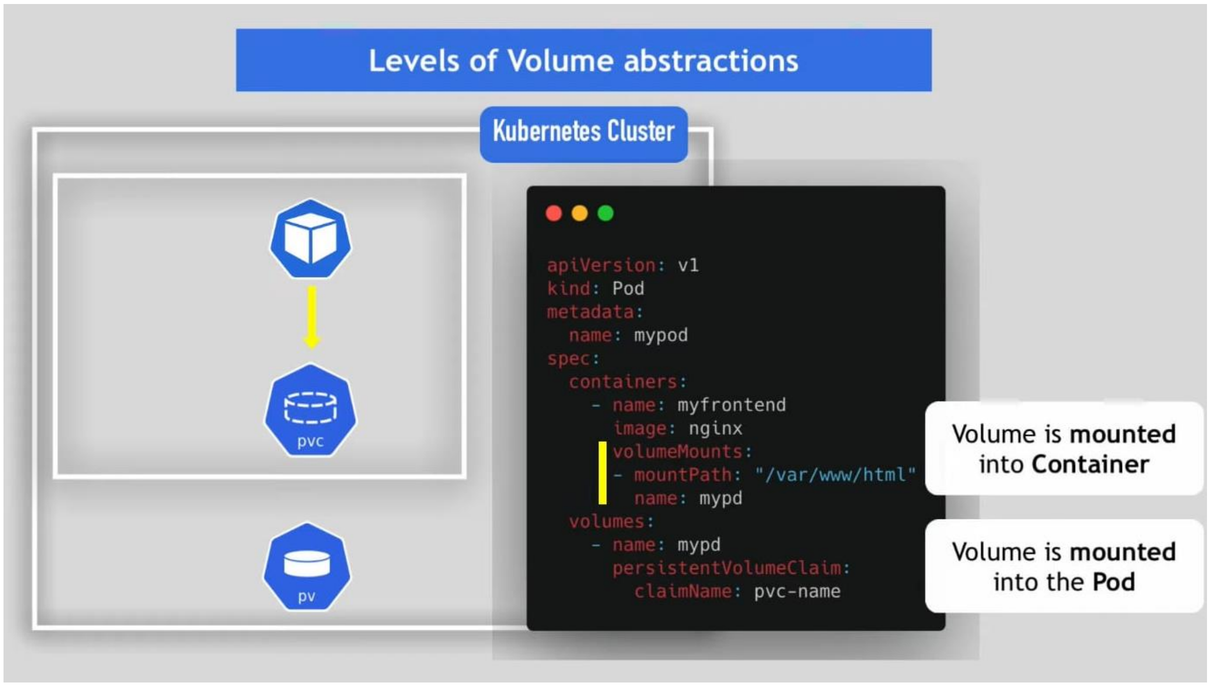
```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc-name
spec:
  storageClassName: manual
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```



Levels of Volume abstractions

Pod requests the volume through the PV claim

Claim tries to find a volume in cluster



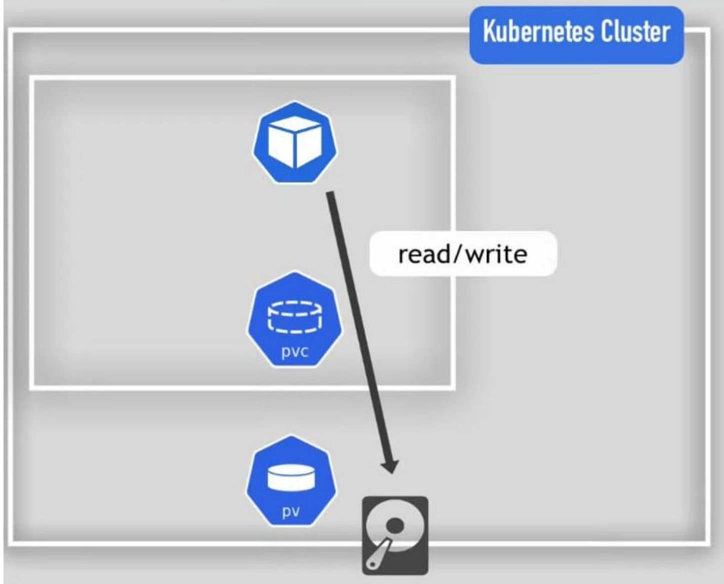
Levels of Volume abstractions

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: pvc-name
```

Volume is **mounted** into **Container**

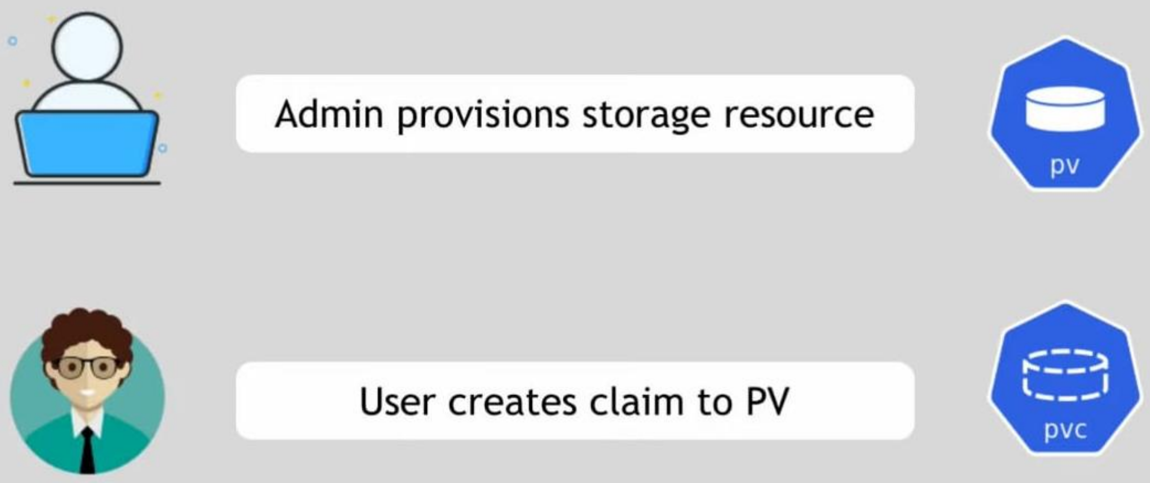
Volume is **mounted** into the **Pod**

Levels of Volume abstractions



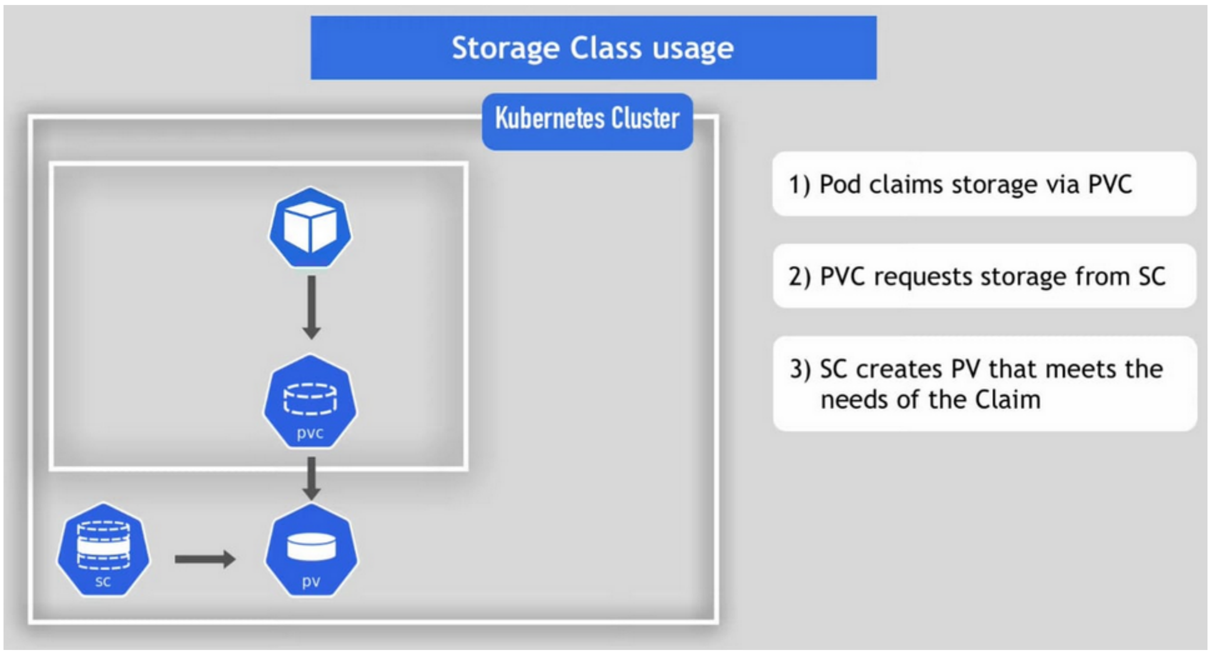
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: pvc-name
```

Why so many abstractions? 🤔

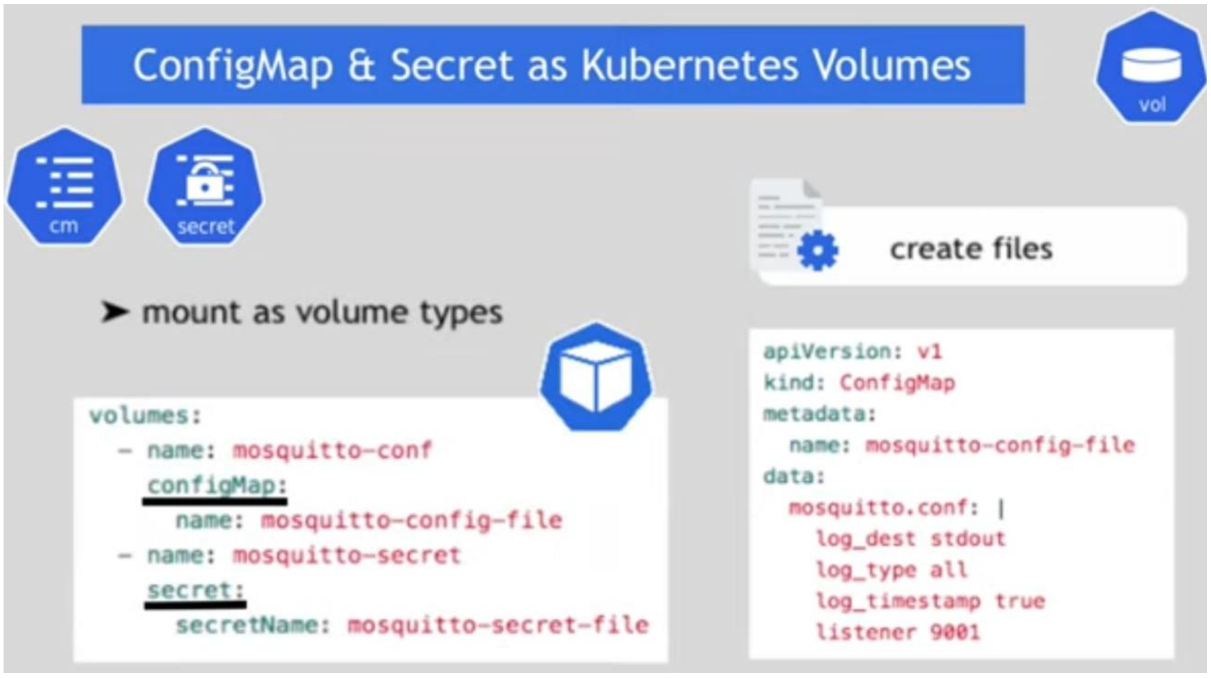


Admin provisions storage resource

User creates claim to PV

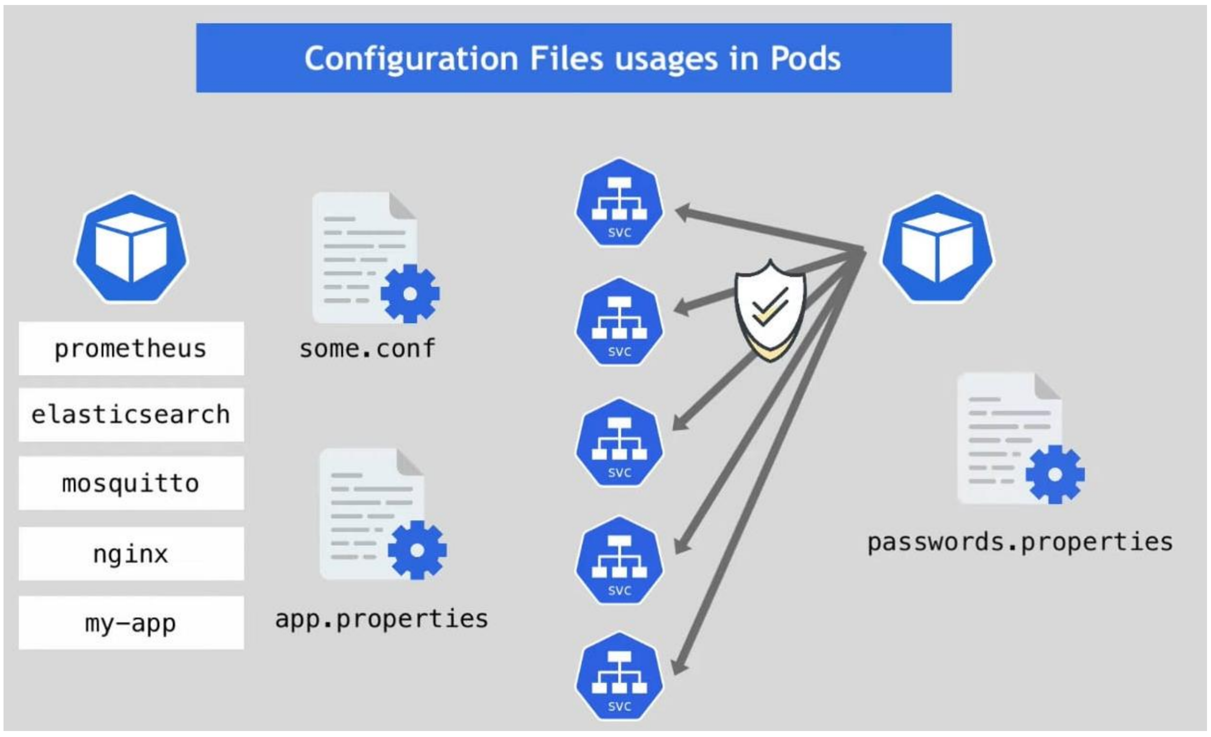

Storage Class usage
Kubernetes Cluster
pvc
sc
pv
1) Pod claims storage via PVC
2) PVC requests storage from SC
3) SC creates PV that meets the needs of the Claim

ConfigMap and Secrete:



```
volumes:
  - name: mosquitto-conf
    configMap:
      name: mosquitto-config-file
  - name: mosquitto-secret
    secret:
      secretName: mosquitto-secret-file
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mosquitto-config-file
data:
  mosquitto.conf: |
    log_dest stdout
    log_type all
    log_timestamp true
    listener 9001
```


Configuration Files usages in Pods
prometheus
elasticsearch
mosquitto
nginx
my–app
some.conf
app.properties
svc
svc
svc
svc
svc
passwords.properties