Kubernetes Storage

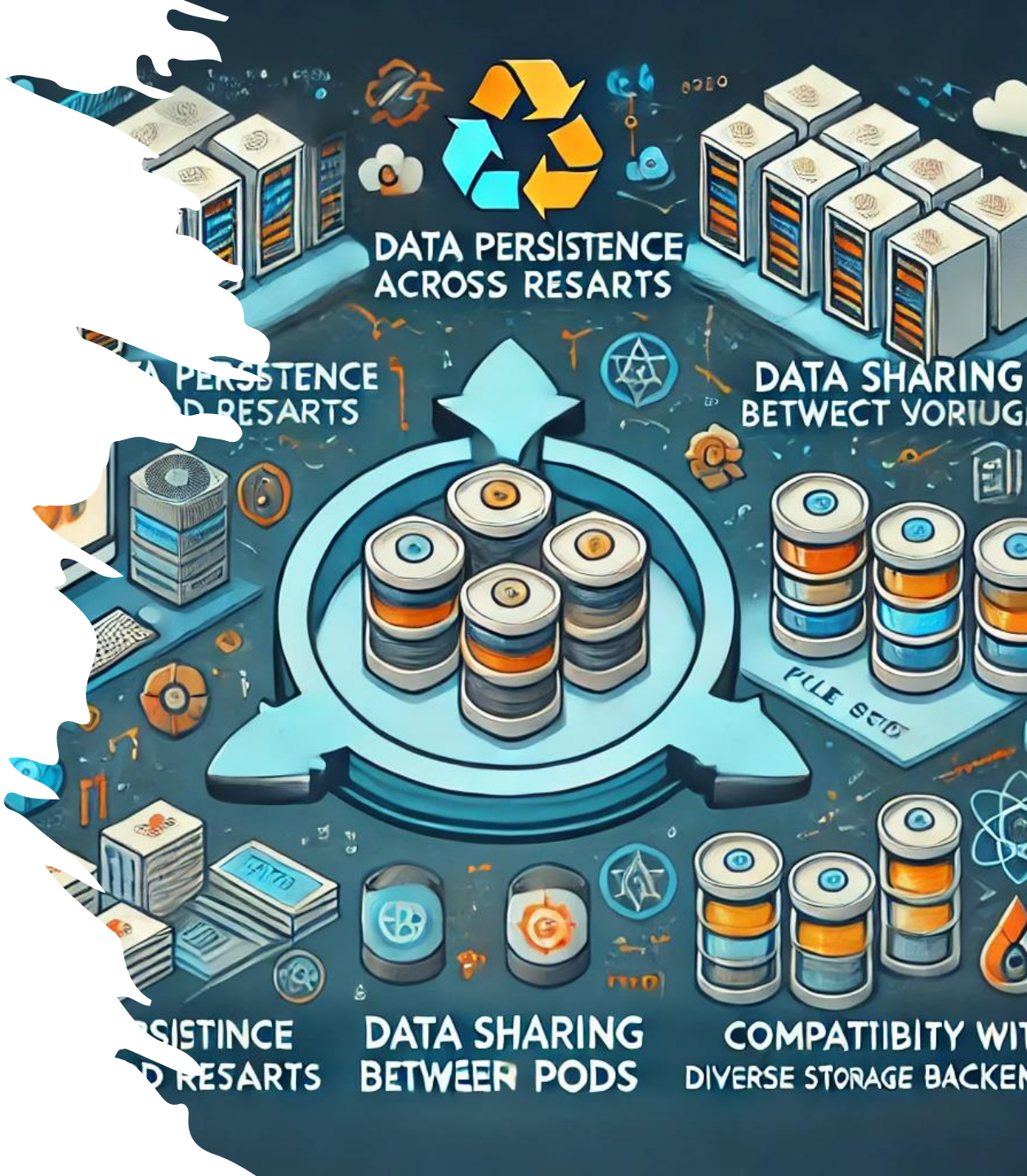
An In-Depth Look

Lisa Zangrando

Kubernetes and Storage: beyond Cloud-Native applications

- **Kubernetes for Cloud-Native applications**
  - designed to manage distributed architectures, dynamic scalability, and resilience.
  - ideal for stateless workloads.

- **The challenge of scientific applications**
  - scientific applications often require **persistent and reliable data access**, especially for use cases like: Data Analysis applications, logging systems, databases

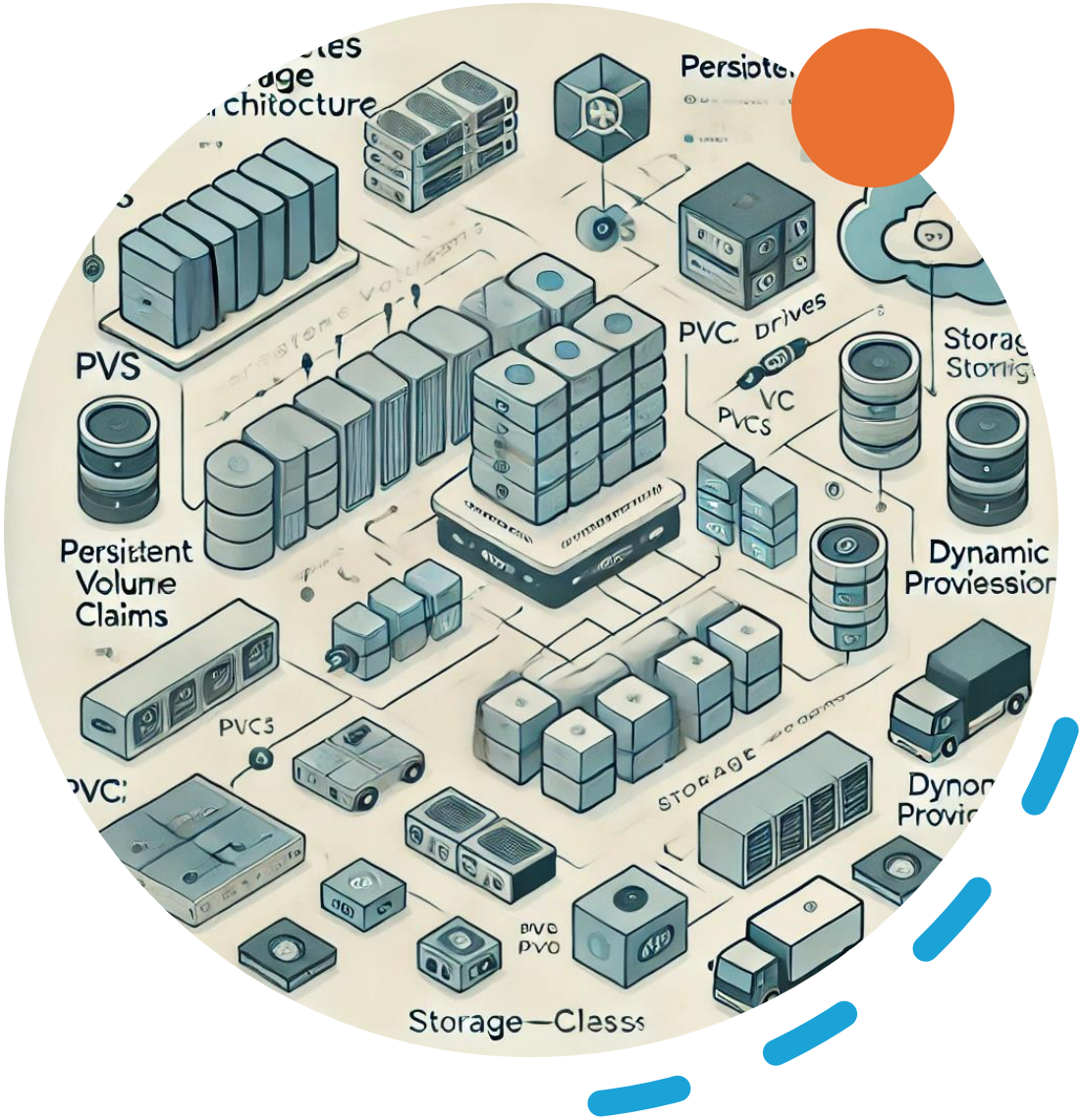


Why is storage indispensable in Kubernetes?

1. **Data persistence across Pod restarts**
   - Pods are **temporary** by design.
   - Without persistent storage, data is **lost** when Pods are deleted or restarted.
   - Persistent storage decouples the **lifecycle of data** from the lifecycle of Pods, ensuring data survives infrastructure changes.

2. **Data sharing between Pods**
   - Distributed applications often require multiple Pods to **simultaneously access** the same data.
   - Kubernetes supports **shared volumes**, enabling efficient and straightforward data sharing.

3. **Compatibility with Diverse Storage Backends**
   - Kubernetes supports a wide range of storage solutions to meet application needs:
     - **Block Storage**: high-performance storage for transactional databases.
     - **Shared File Systems**: collaborative or distributed applications.
     - **Object Storage**: scalable, long-term storage for big data systems.



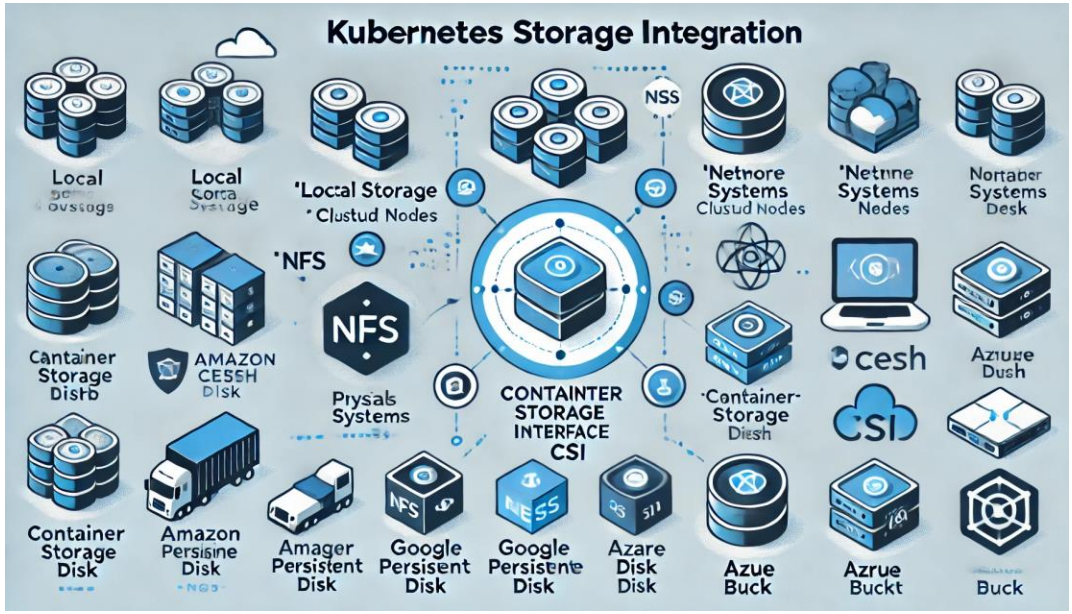
Storage integration in Kubernetes

- Kubernetes simplifies storage management by providing native integrations with various types of infrastructures:
  - **Local storage: d**irectly using physical disks attached to cluster nodes.
  - **Network systems:** solutions like **NFS** or **Ceph**, offering shared access and scalability.
  - **Cloud-Native services:** options such as **Amazon EBS**, **Google Persistent Disk**, **Azure Disk**, and S3-like object storage services.

- Kubernetes supports custom storage plugins through the **Container Storage Interface (CSI)**.

- CSI allows developers to integrate **any storage system**:
  - **Commercial** solutions
  - **Custom** setups

- This makes Kubernetes a **universal solution** for:
  - **On-Premises**
  - **Hybrid**
  - **Cloud-Native Environments**



A flexible architecture based on Volumes

- **The core of Kubernetes storage: Volumes**
  - Volumes are abstractions that allow containers to access storage resources without being dependent on the underlying infrastructure.
  - A Kubernetes Volume is essentially a directory mounted into containers within a Pod.

- **Simplified data handling for applications**
  - Volumes provide Pods with a mechanism to **read and write files**, abstracting the complexities of storage backend connections.
  - Containers remain unaware of the complexity of the underlying storage.

- **Volume usage in Pods**
  - Once created, a Volume is **mounted into containers** as a directory, becoming an integral part of the container's filesystem.
  - Enables **data sharing between containers** in the same Pod:
    - Ideal for multi-container applications where one container generates data for another.

How Volumes work?

A volume is defined in the **pod spec.**

Volumes are **mounted** into containers, making them shared and accessible at specified paths.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: busybox
    volumeMounts:
    - mountPath: /data
      name: my-volume-1
    - mountPath: /app
      name: my-volume-2
  volumes:
  - name: my-volume-1
    <VOLUME-DEFINITON>
  - name: my-volume-2
    <VOLUME-DEFINITON>
```

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container-1
    image: busybox
    volumeMounts:
    - mountPath: /data
      name: my-volume
- name: my-container-2
    image: busybox
    volumeMounts:
    - mountPath: /storage
      name: my-volume
  volumes:
  - name: my-volume
    <VOLUME-DEFINITON>
```

Types of Volumes

- **Ephemeral Volumes:**
  - Temporary and tied to the **lifecycle of the Pod**.
  - Commonly used for caching, temporary data, or inter-container communication.

- **Persistent Volumes (PVs):**
  - Designed for long-term storage, independent of a Pod's lifecycle.
  - Ideal for applications requiring **durable storage**, such as databases.
  - Each type of volume serves distinct use cases, addressing different levels of **data persistence** and lifecycle requirements.

Ephemeral Volumes

- **Definition**:
  - Ephemeral volumes are temporary storage solutions tied to the lifecycle of a pod. They are created when the pod starts and are deleted when the pod is terminated.
- **Use cases**:
  - **Scratch space**: temporary storage for processing data within a single pod lifecycle.
  - **Caching**: speed up operations with local, short-term storage.
  - **Config and Secrets**: lightweight storage for sensitive or temporary data.
- **Types**:
  - **EmptyDir**: storage created on the node's disk or memory, shared among containers in the pod.
  - **ConfigMap and Secret**: store configuration files or sensitive data.
  - **CSI Ephemeral Volumes**: temporary storage from a CSI driver.

```
apiVersion: v1
kind: Pod
metadata:
   name: ephemeral-example
spec:
  containers:
  - name: app-container
    image: nginx
    volumeMounts:
    - mountPath: "/usr/share/nginx/html"
      name: scratch-volume
  volumes:
  - name: scratch-volume
    emptyDir: {}
```

Explanation:

- The `emptyDir` volume is created when the pod starts.
- It mounts at `/usr/share/nginx/html` in the `app-container`.
- The volume is deleted when the pod is terminated.

Ephemeral Volumes
(ConfigMap example)

- **Definition**:
  - ConfigMap is a type of ephemeral volume used to provide configuration data, such as environment variables or configuration files, to containers.
- **Key Features**:
  - Allows decoupling configuration from container images.
  - Automatically updates when the ConfigMap changes (depending on pod settings).
- **Example**:
  - The ConfigMap example-config contains a simple HTML file.
  - The pod mounts the ConfigMap as a volume (config-volume) at /usr/share/nginx/html.
  - When the pod runs, the NGINX server serves the content of the index.html file from the ConfigMap.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-config
data:
  index.html: |
    <html>
      <head><title>Welcome</title></head>
      <body><h1>Hello, Kubernetes!</h1></body>
    </html>

---

apiVersion: v1
kind: Pod
metadata:
  name: configmap-example
spec:
  containers:
  - name: app-container
    image: nginx
    volumeMounts:
    - mountPath: "/usr/share/nginx/html"
      name: config-volume
  volumes:
  - name: config-volume
    configMap:
      name: example-config
```

Persistent Volumes

- **Definition**:
  - A Persistent Volume (PV) is a cluster-wide storage resource provisioned independently of any specific pod. It enables data persistence beyond the lifecycle of pods.
- **Key Features**:
  - Decouples storage from pod lifecycle.
  - Supports multiple backends (e.g., NFS, Ceph, AWS EBS, GCE Persistent Disk).
  - **Provisioned:**
    - **statically (manually) by the administrator**
    - **dynamically by a storage provider via StorageClass defined by the adminisrtator.**
- **Components**:
  - **PersistentVolume (PV)**: Represents the physical storage resource.
  - **PersistentVolumeClaim (PVC)**: A request for storage by a pod.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: static-pv
spec:
  capacity:
    storage: 1Gi
    accessModes:
    - ReadWriteOncePod
    hostPath:
      path: /data/static-pv
```

- A static `hostPath` volume is created, with 1Gi of storage and access mode **`ReadWriteOncePod`**.
- The volume is created on a node at the path `/data/static-pv`.

Persistent Volume Claim

- **Definition**:
  - A **PersistentVolumeClaim (PVC)** is a request for storage resources in Kubernetes.
  - It is used by pods to request and consume persistent storage that has been provisioned via a **PersistentVolume (PV)**.
  - PVCs enable dynamic or static provisioning of storage resources in Kubernetes.
- **How PVC works**
  - **Create PVC**: a user or pod creates a PVC that specifies the desired storage capacity, access mode, and other parameters (such as storage class).
  - **Binding**: the Kubernetes control plane looks for an available PV that matches the request in the PVC. If a matching PV is found, the PVC is bound to it.
  - **Pod usage**: once bound, the PVC can be used by one or more pods to mount the volume and access its data.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: static-pvc
spec:
  accessModes:
  - ReadWriteOncePod
  resources:
    requests:
      storage: 1Gi
```

- **PersistentVolumeClaim**:
  - A PVC requests 1Gi of storage with **`ReadWriteOncePod`** access.
  - Kubernetes will try to find a PV that satisfies the claim's requirements (1Gi storage and access mode `ReadWriteOncePod`).
- **Binding**:
  - The PVC will **bind** to the `static-pv` if it matches the criteria.
  - Once bound, the volume is ready for use by a pod.

```
# kubectl get pv static-pv
NAME        CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
static-pv   1Gi        RWOP           Retain           Bound    static-
pvc

k get pvc static-pvc
NAME         STATUS   VOLUME      CAPACITY   ACCESS MODES
static-pvc   Bound    static-pv   1Gi        RWOP
```

Using PVC in a Pod

- A pod can then use the PVC to mount the volume
- **Key Points to Remember**:
  - **PVCs abstract storage management**: PVCs allow users to request storage without needing to understand the specifics of the underlying infrastructure (such as whether the volume is hostPath, NFS, or cloud storage).
  - **Binding**: PVCs are bound to PVs based on criteria such as access modes and storage capacity.
  - **Access Modes**: a PVC with ReadWriteOncePod can only mount the volume in read-write mode on one pod at a time.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: container1
    image: nginx
    volumeMounts:
    - mountPath: "/usr/share/nginx/html"
      name: storage
  volumes:
  - name: storage
    persistentVolumeClaim:
      claimName: static-pvc
```

- The **`claimName: static-pvc`** in the pod's `volumes` section tells Kubernetes to mount the volume that is bound to the `static-pvc` claim.
- The pod can now access the `/data` path on the PV (`/data/static-pv` on the node).

StorageClass
(dynamic provisioning)

- **Definition**:
  - A **StorageClass** is a Kubernetes abstraction that defines the characteristics of **dynamic storage**.
  - It allows administrators to specify storage configurations (e.g. type, performance, and policies) to automate the creation of Persistent Volumes
- **Why is it important?**
  - **Dynamic provisioning:** automates the PV creation process when an application requests storage via a PersistentVolumeClaim.
  - **Flexibility:** provides predefined "profiles" of storage (e.g., high-speed, cost-effective, or highly available) for developers to use without worrying about infrastructure details.
  - **Centralized management:** enables administrators to configure StorageClasses to meet diverse application needs while retaining resource management control.

How does a StorageClass work?

1. **Definition:** administrator create a StorageClass, specifying a storage driver and its options.
2. **Request via PVC:** A PersistentVolumeClaim (PVC) specifies a StorageClass.
3. **Dynamic Provisioning:** Kubernetes uses the driver in the StorageClass to automatically create a PV that matches the PVC's requirements.
4. **Binding:** The created PV is automatically bound to the PVC.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs-csi
provisioner: nfs.csi.k8s.io
parameters:
  server: 192.168.81.177
  share: /srv/nfs_share

---

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: nfs-csi
```

Types of Persistent Volumes

- Kubernetes supports various types of Persistent Volumes, each with different characteristics and use cases.
- Kubernetes manages the two kind of storage plugins (drivers) that handle these PVs: **in-tree** and **out-of-tree**
- **In-tree drivers**
  - Integrated directly into Kubernetes' source code.
  - Each driver is part of the core system, meaning updates and modifications depend on Kubernetes release cycles.
  - **Examples:** awsElasticBlockStore, gcePersistentDisk, azureDisk.
- **Out-of-tree drivers**
  - Implemented externally to Kubernetes' source code.
  - Built using the modular, flexible and standardize **CSI (Container Storage Interface)** architecture.
  - Updates independent of Kubernetes release cycles.

Summary of PV types:

PV Type	Access Modes	Use Case
hostPath	ReadWriteOnce	Local storage on a specific node
NFS	ReadWriteMany	Shared storage accessible by multiple nodes
iSCSI	ReadWriteOnce	High-performance block storage over the network
CephFS	ReadWriteMany	Distributed file system with high availability
AWS EBS	ReadWriteOnce	Persistent block storage on AWS
Azure Disk	ReadWriteOnce	Persistent block storage on Azure
Google Cloud Persistent Disk	ReadWriteOnce	Persistent block storage on Google Cloud
CSI	Depends on provider	External storage solutions integrated with Kubernetes

Thanks!

References

- https://kubernetes.io/docs/concepts/storage/volumes/
- https://kubernetes.io/docs/concepts/storage/persistent-volumes/
- https://kubernetes.io/docs/concepts/storage/ephemeral-volumes/
- https://kubernetes.io/docs/concepts/storage/storage-classes/
- https://kubernetes.io/docs/concepts/storage/dynamic-provisioning/
- https://medium.com/geekculture/storage-kubernetes-92eb3d027282
- https://medium.com/@martin.hodges/adding-persistent-storage-to-your-kubernetes-cluster-5e12adb81592