Docker

Manage Application Data

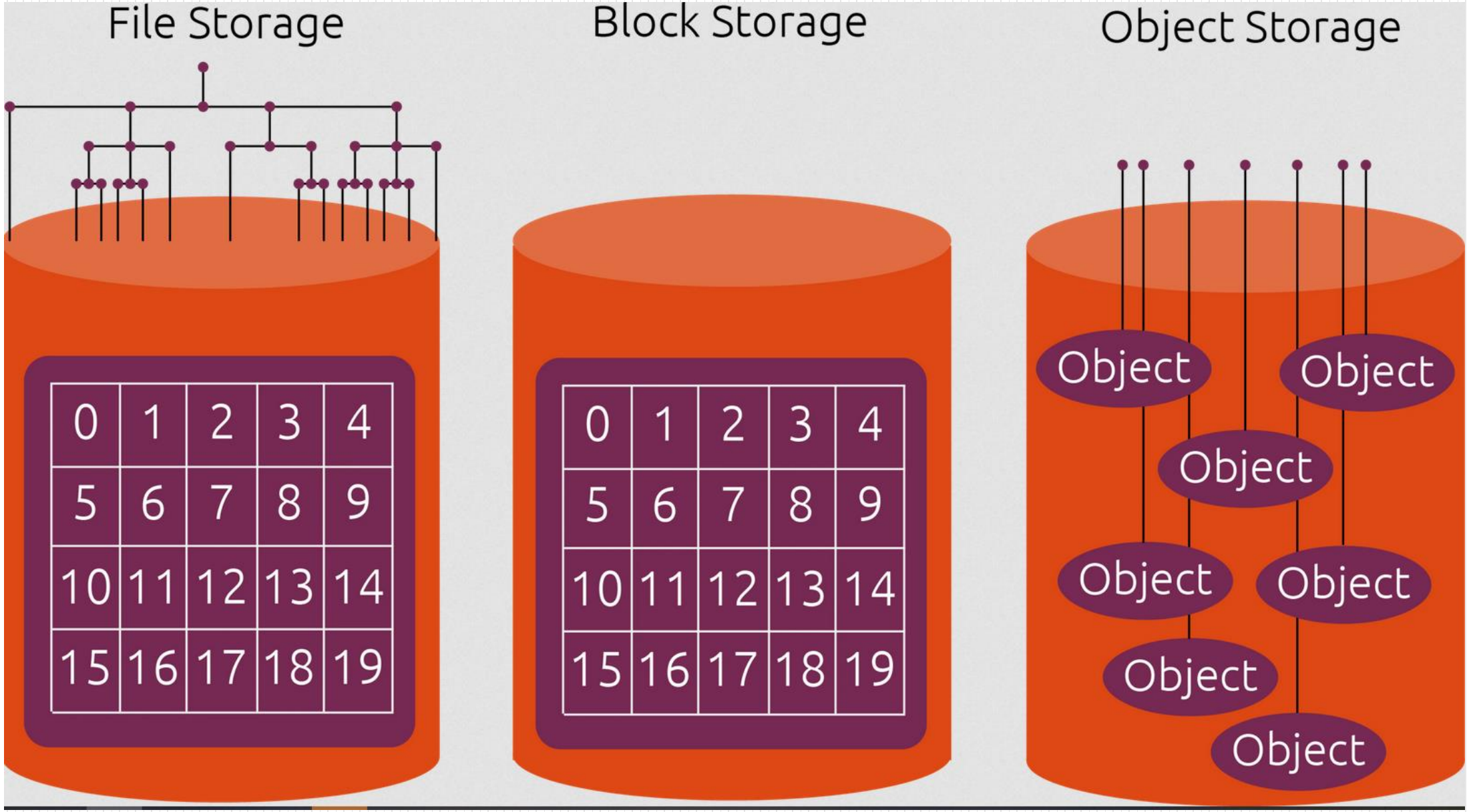
Docker Application Data

Linux storage overview

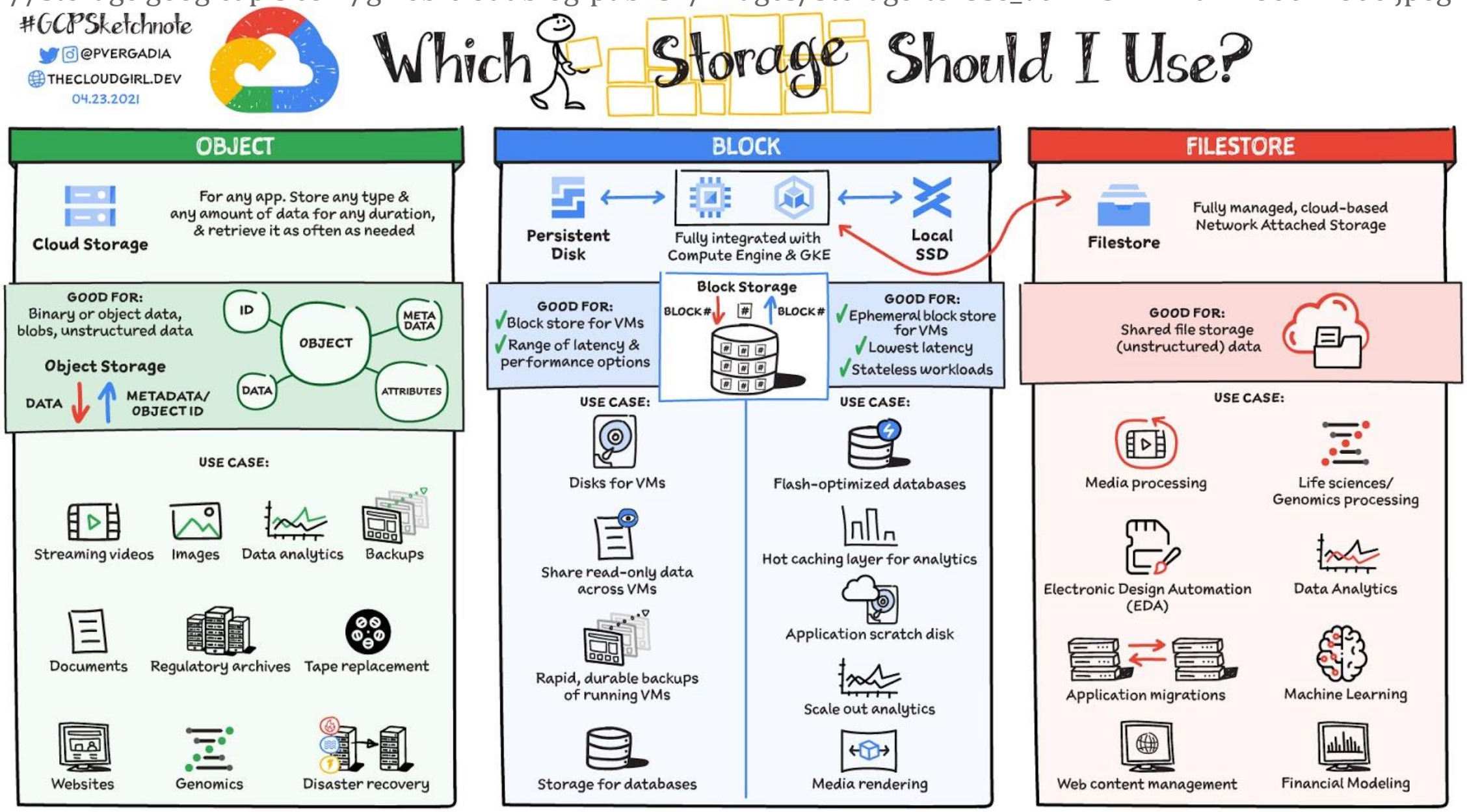
Linux Storage concepts

- Linux Storage building blocks:
  - Level 0: Block device (SATA, SAS, NVMe, USB, ISCSI, FC, FCoE ….)
  - Level 1: Disk partitions (MBR, GPT) (optional)
  - Level 2: Volume managers (LVM, ZFS, Btrfs) (optional)
  - Level 3: Filesystems (Ext, Btrfs, ZFS, GFS …)

https://ubuntu.com/blog/what-are-the-different-types-of-storage-block-object-and-file



https://storage.googleapis.com/gweb-cloudblog-publish/images/Storage-to-Use_v04-23-21.max-1600x1600.jpeg



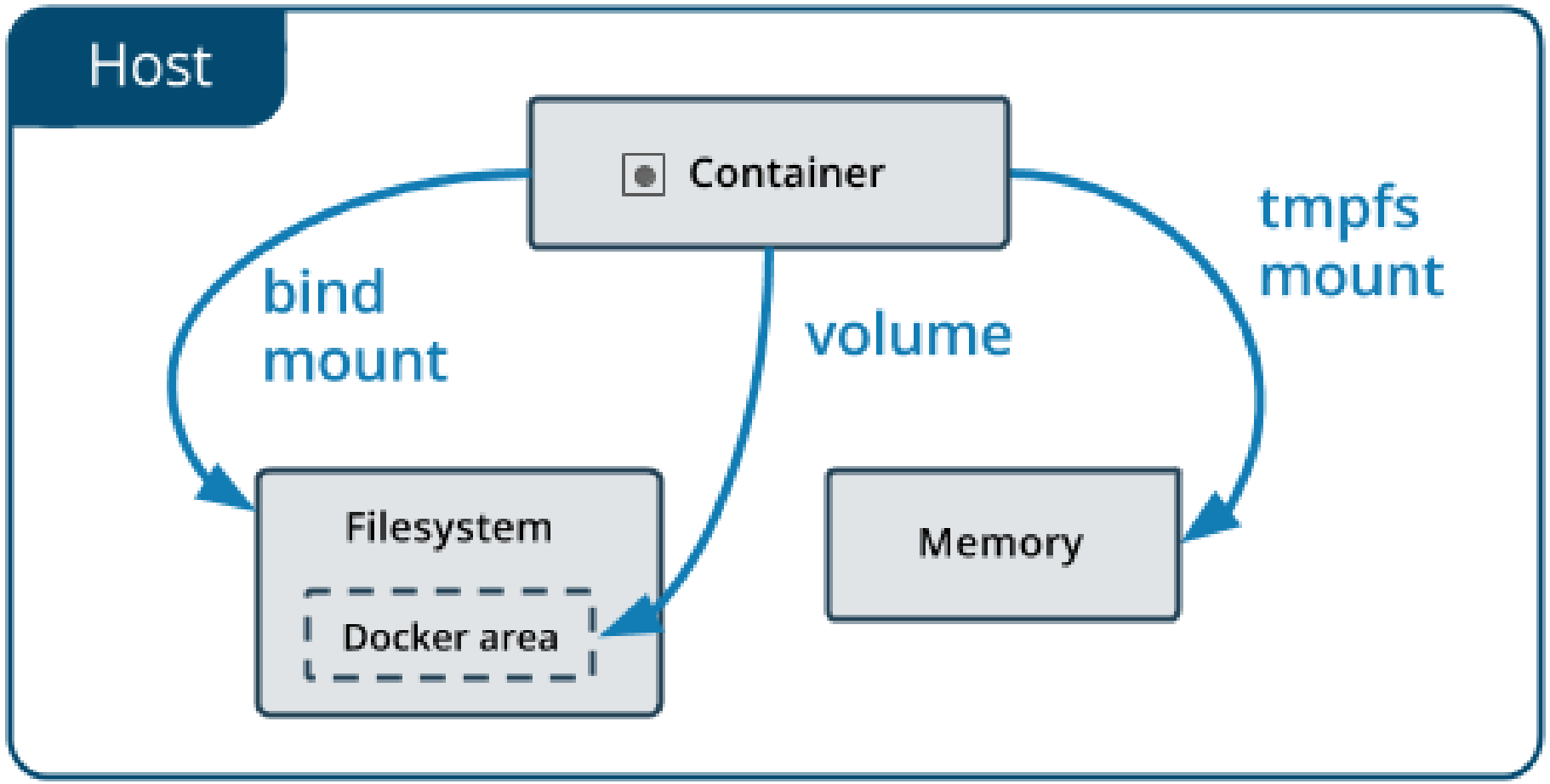
Docker Application Data

Application Data Concepts

Choose the right type of mount

- No matter which type of mount you choose to use, the data looks the same from within the container. It is exposed as either a directory or an individual file in the container's filesystem.
- **Volumes** are stored in a part of the host filesystem which is managed by Docker (/var/lib/docker/volumes/ on Linux). Non-Docker processes should not modify this part of the filesystem. Volumes are the best way to persist data in Docker.
- **Bind mounts** may be stored anywhere on the host system. They may even be important system files or directories. Non-Docker processes on the Docker host or a Docker container can modify them at any time.
- **tmpfs mounts** are stored in the host system's memory only, and are never written to the host system's filesystem.

Docker mount points



Use cases for volumes

- Sharing data among multiple running containers. If you don't explicitly create it, a volume is created the first time it is mounted into a container. When that container stops or is removed, the volume still exists. Multiple containers can mount the same volume simultaneously, either read-write or read-only. Volumes are only removed when you explicitly remove them.
- When the Docker host is not guaranteed to have a given directory or file structure. Volumes help you decouple the configuration of the Docker host from the container runtime.
- When you want to store your container's data on a remote host or a cloud provider, rather than locally.
- When you need to back up, restore, or migrate data from one Docker host to another, volumes are a better choice. You can stop containers using the volume, then back up the volume's directory

Use cases for bind mounts

- Sharing configuration files from the host machine to containers. This is how Docker provides DNS resolution to containers by default, by mounting /etc/resolv.conf from the host machine into each container.
- Sharing source code or build artifacts between a development environment on the Docker host and a container. For instance, you may mount a Maven target/ directory into a container, and each time you build the Maven project on the Docker host, the container gets access to the rebuilt artifacts.
- If you use Docker for development this way, your production Dockerfile would copy the production-ready artifacts directly into the image, rather than relying on a bind mount.
- When the file or directory structure of the Docker host is guaranteed to be consistent with the bind mounts the containers require.

Use cases for tmpfs mounts

- tmpfs mounts are best used for cases when you do not want the data to persist either on the host machine or within the container. This may be for security reasons or to protect the performance of the container when your application needs to write a large volume of non-persistent state data.

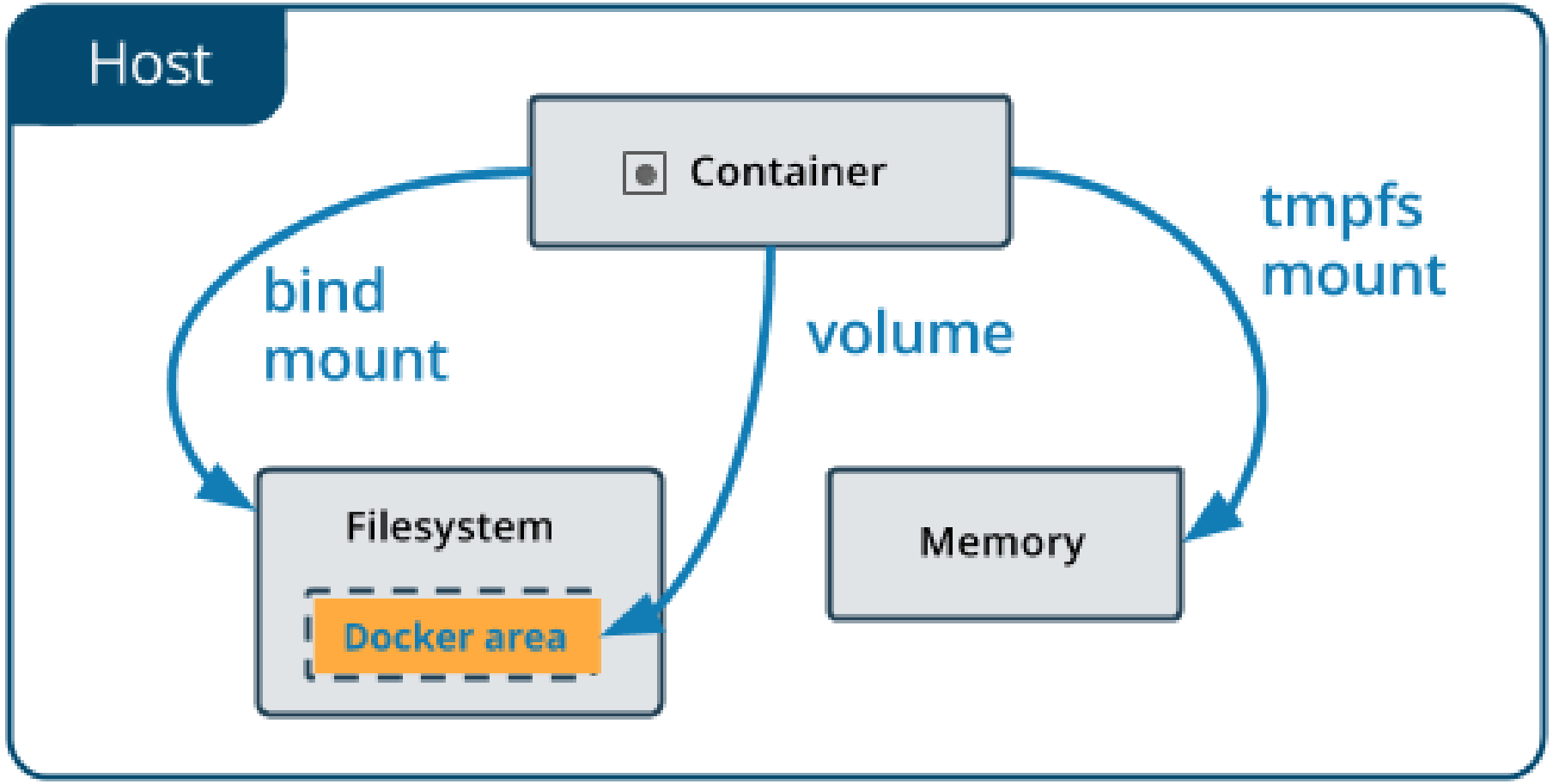
Docker Application Data

Volumes

Volumes

- Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:
  - Volumes are easier to back up or migrate than bind mounts.
  - You can manage volumes using Docker CLI commands or the Docker API.
  - Volumes work on both Linux and Windows containers.
  - Volumes can be more safely shared among multiple containers.
  - Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
  - New volumes can have their content pre-populated by a container.
  - Volumes don’t increase the size of the containers using it, and the volume’s contents exist outside the lifecycle of a given container.

Volumes


Host
Container
bind
mount
volume
tmpfs
mount
Filesystem
Docker area
Memory

Create a volume

- Unlike a bind mount, you can create and manage volumes outside the scope of any container.

- Create a volume:

```
docker volume create my-vol
```

- List volumes:

```
docker volume ls
local               my-vol
```

Inspect a volume

- Inspect a volume:

```
docker volume inspect my-vol
[
    {
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/my-vol/_data",
        "Name": "my-vol",
        "Options": {},
        "Scope": "local"
    }
]
```

Start a container with a volume

- If you start a container with a volume that does not yet exist, Docker creates the volume for you.
- Using --mount syntax

```
docker run -d  --name devtest  --mount source=myvol2,target=/app \
      nginx:latest
```

- Using --volume (-v) syntax

```
docker run -d  --name devtest \
      -v myvol2:/app nginx:latest
```

Choose the -v or --mount flag

- Originally, the -v or --volume flag was used for standalone containers and the --mount flag was used for swarm services.
- However, starting with Docker 17.06, you can also use --mount with standalone containers.
- The biggest difference is that the -v syntax combines all the options together in one field, while the --mount syntax separates them.
- When using volumes with services, only --mount is supported.

Remove a volume

- Stop the container and remove the volume. Note volume removal is a separate step.

```
docker container stop devtest

docker container rm devtest

docker volume rm myvol2
```

Populate a volume using a container

- If you start a container which creates a new volume and the container has files or directories in the directory to be mounted (such as /app/ above), the directory's contents are copied into the volume. The container then mounts and uses the volume, and other containers which use the volume also have access to the pre-populated content.

```
docker run -d  --name=nginxtest  -v nginx-vol:/usr/share/nginx/html \
 nginx:latest

docker run -d  --name=nginxtest \
 --mount source=nginx-vol,destination=/usr/share/nginx/html \
 nginx:latest
```

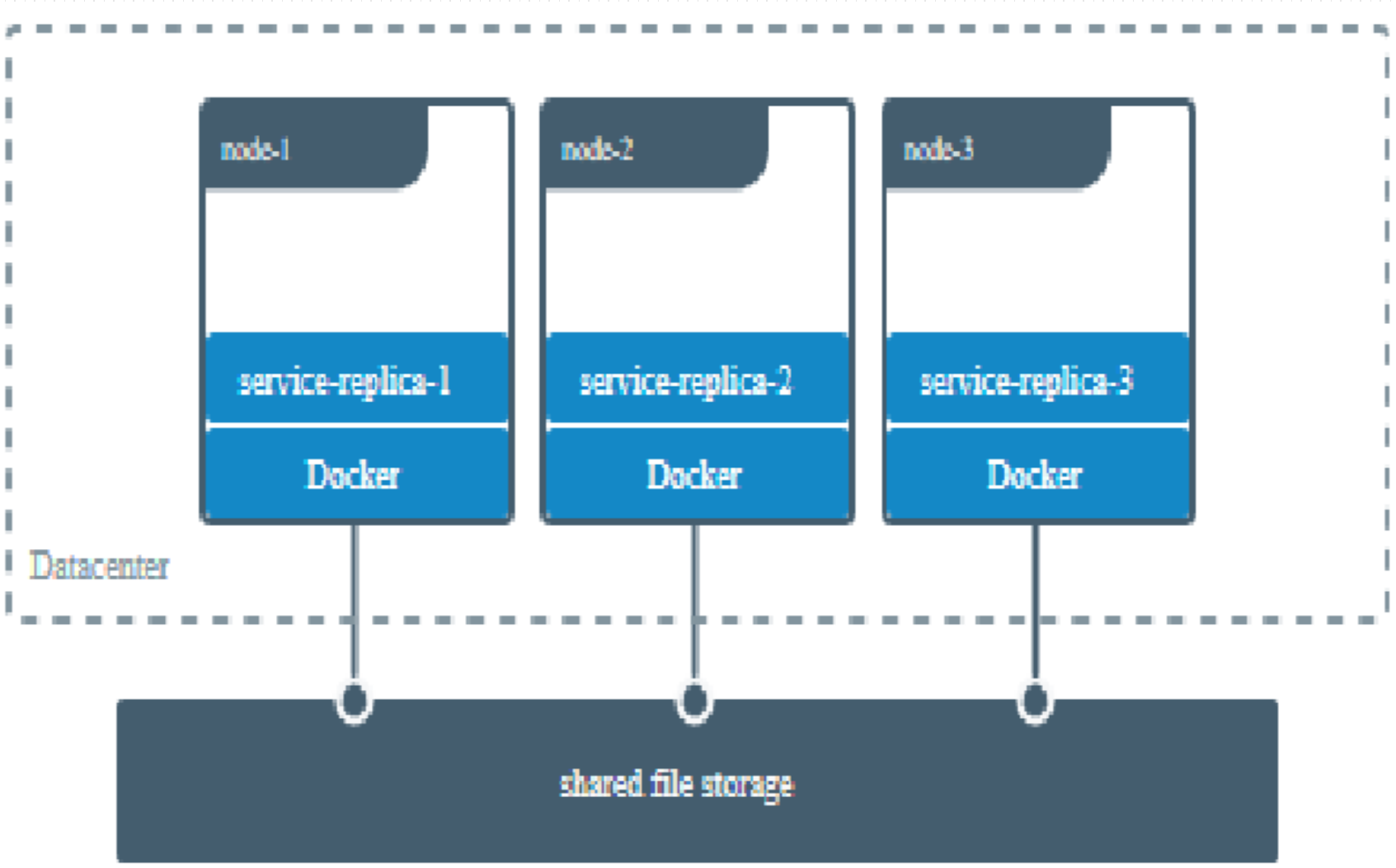
Use a read-only volume

- For some development applications, the container needs to write into the bind mount so that changes are propagated back to the Docker host. At other times, the container only needs read access to the data. Remember that multiple containers can mount the same volume, and it can be mounted read-write for some of them and read-only for others, at the same time.

```
docker run -d --name=nginxtest \
 --mount source=nginx-vol,destination=/usr/share/nginx/html,readonly \
 nginx:latest

docker run -d  --name=nginxtest   -v nginx-vol:/usr/share/nginx/html:ro \
 nginx:latest
```

Share data among machines



Initial set-up

- This example assumes that you have two nodes, the first of which is a Docker host and can connect to the second using SSH.

- On the Docker host, install the vieux/sshfs plugin:

```
docker plugin install --grant-all-permissions vieux/sshfs
```

Create a volume using a volume driver

- This example specifies a SSH password, but if the two hosts have shared keys configured, you can omit the password. Each volume driver may have zero or more configurable options, each of which is specified using an -o flag.

```
docker volume create --driver vieux/sshfs \
 -o sshcmd=test@node2:/home/test \
 -o password=testpassword \
 sshvolume
```

Start a container which creates a volume using a volume driver

- This example specifies a SSH password, but if the two hosts have shared keys configured, you can omit the password. Each volume driver may have zero or more configurable options. If the volume driver requires you to pass options, you must use the --mount flag to mount the volume, rather than -v.

```
docker run -d --name sshfs-container  --volume-driver vieux/sshfs \
  --mount src=sshvolume,target=/app,volume-
opt=sshcmd=test@node2:/home/test,volume-opt=password=testpassword \
  nginx:latest
```

Create a service which creates an NFS volume

- This example shows how you can create an NFS volume when creating a service. This example uses 10.0.0.10 as the NFS server and /var/docker-nfs as the exported directory on the NFS server.
- **Note** that the volume driver specified is local.
- NFSv3

```
docker service create -d   --name nfs-service \
  --mount 'type=volume,source=nfsvolume,target=/app,volume-
driver=local,volume-opt=type=nfs,volume-opt=device=:/var/docker-nfs,volume-
opt=o=addr=10.0.0.10' \
  nginx:latest
```

Create a service which creates an NFS volume

- This example shows how you can create an NFS volume when creating a service. This example uses 10.0.0.10 as the NFS server and /var/docker-nfs as the exported directory on the NFS server.
- **Note** that the volume driver specified is local.
- NFSv4

```
docker service create -d  --name nfs-service \
   --mount 'type=volume,source=nfsvolume,target=/app,volume-
driver=local,volume-opt=type=nfs,volume-opt=device=:/,"volume-
opt=o=10.0.0.10,rw,nfsvers=4,async"' \
   nginx:latest
```

Backup a container

- In the next command, we:
- Launch a new container and mount the volume from the dbstore container
- Mount a local host directory as /backup
- Pass a command that tars the contents of the dbdata volume to a backup.tar file inside our /backup directory.

```
docker run --rm --volumes-from dbstore\
 -v $(pwd):/backup ubuntu tar cvf /backup/backup.tar /dbdata
```

Restore a container

- With the backup just created, you can restore it to the same container, or another that you made elsewhere.
- For example, create a new container named dbstore2:

```
docker run -v /dbdata --name dbstore2 ubuntu /bin/bash
```

- Then un-tar the backup file in the new container`s data volume:

```
docker run --rm --volumes-from dbstore2 \
-v $(pwd):/backup ubuntu  \
bash -c "cd /dbdata && tar xvf /backup/backup.tar --strip 1"
```

Remove volumes

- A Docker data volume persists after a container is deleted. There are two types of volumes to consider:

- Named volumes have a specific source from outside the container, for example awesome:/bar.
- Anonymous volumes have no specific source so when the container is deleted, instruct the Docker Engine daemon to remove them.

Remove all volumes

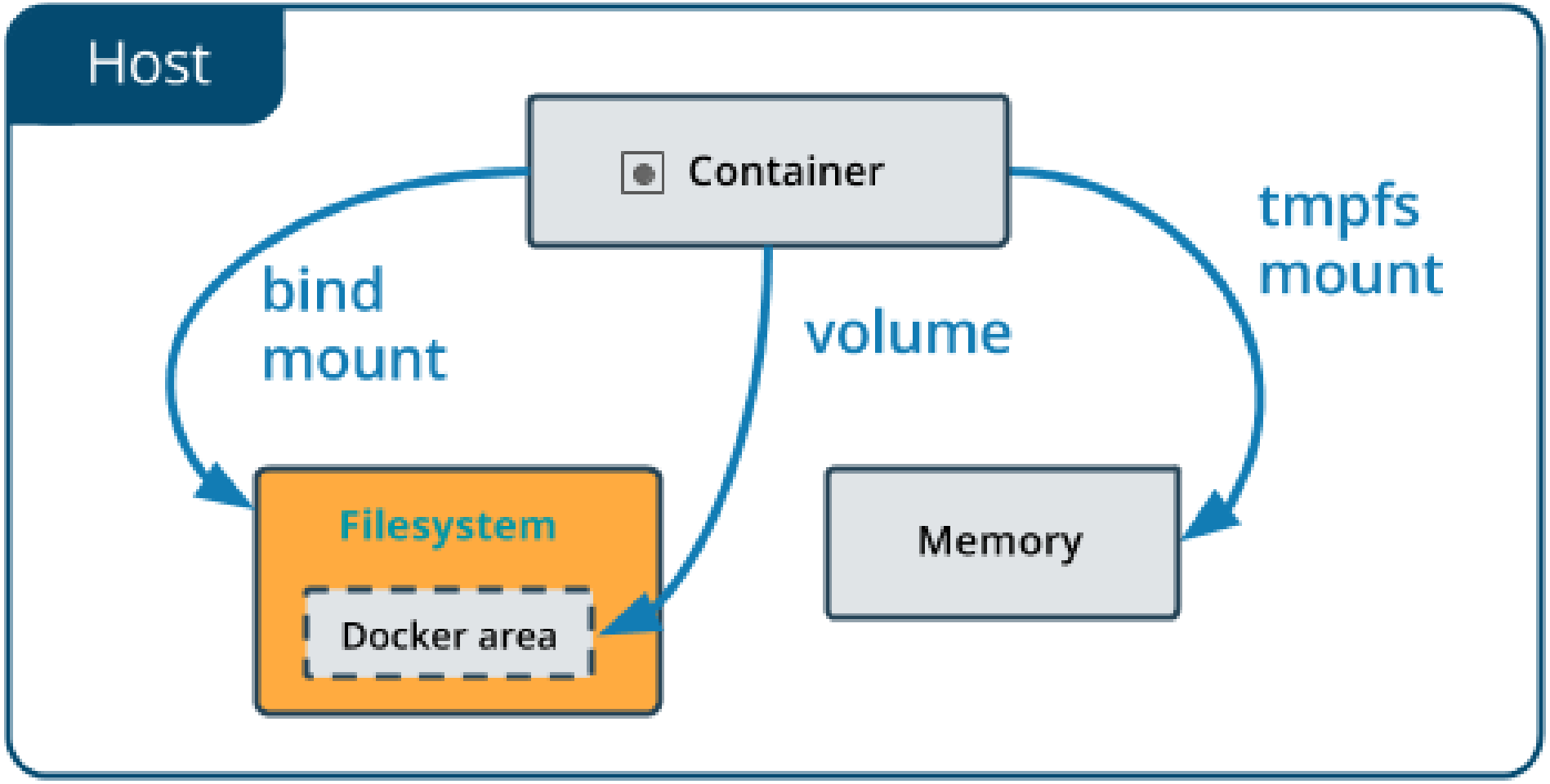
- To remove all unused volumes and free up space:

```
docker volume prune
```

Docker Application Data

Bind mounts

Bind mounts


Host
Container
bind
mount
volume
tmpfs
mount
Filesystem
Docker area
Memory

Differences between -v and --mount behavior

- Because the -v and --volume flags have been a part of Docker for a long time, their behavior cannot be changed. This means that there is one behavior that is different between -v and --mount.
- If you use -v or --volume to bind-mount a file or directory that does not yet exist on the Docker host, -v creates the endpoint for you. It is always created as a directory.
- If you use --mount to bind-mount a file or directory that does not yet exist on the Docker host, Docker does not automatically create it for you, but generates an error.

Start a container with a bind mount

- Use the following command to bind-mount the target/ directory into your container at /app/. Run the command from within the source directory. The $(pwd) sub-command expands to the current working directory on Linux or macOS hosts.

```
docker run -d -it --name devtest \
        -v "$(pwd)"/target:/app nginx:latest

docker run -d  -it  --name devtest \
        --mount type=bind,source="$(pwd)"/target,target=/app \
         nginx:latest
```

Start a container with a bind mount

- Use the following command to bind-mount the target/ directory into your container at /app/. Run the command from within the source directory. The $(pwd) sub-command expands to the current working directory on Linux or macOS hosts.

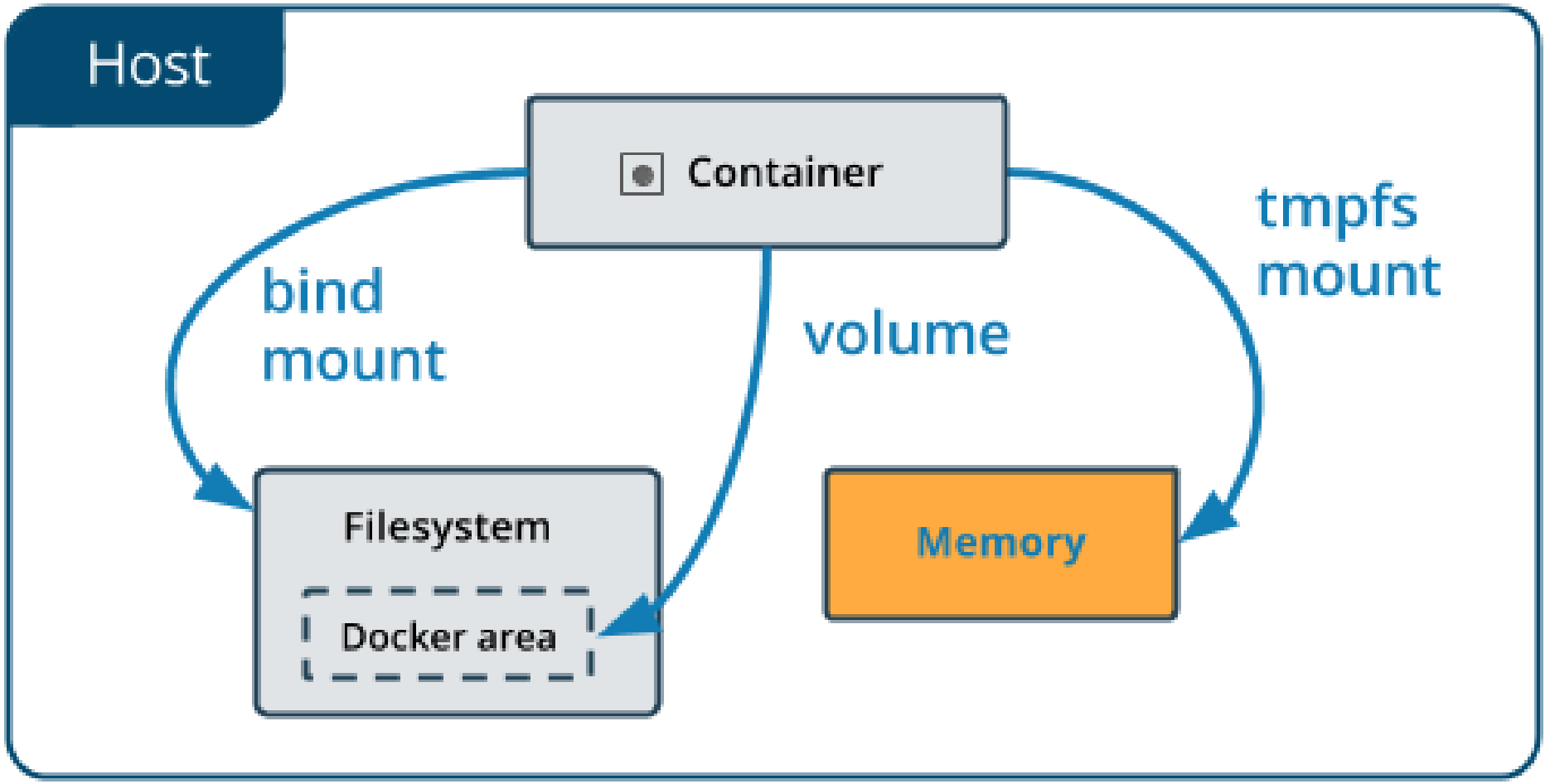
```
docker run -d -it --name devtest \
        -v "$(pwd)"/target:/app nginx:latest

docker run -d  -it  --name devtest \
        --mount type=bind,source="$(pwd)"/target,target=/app \
         nginx:latest
```

Docker Application Data

tmpfs mounts

tmpfs mounts



tmpfs mounts

- Differences between --tmpfs and --mount behavior

- The --tmpfs flag does not allow you to specify any configurable options.
- The --tmpfs flag cannot be used with swarm services. You must use --mount.

Use a tmpfs mount in a container

- To use a tmpfs mount in a container, use the --tmpfs flag, or use the --mount flag with type=tmpfs and destination options. There is no source for tmpfs mounts.

```
docker run -d -it --name tmptest \
        --mount type=tmpfs,destination=/app nginx:latest

docker run -d -it --name tmptest \
        --tmpfs /app   nginx:latest
```

Specify tmpfs options

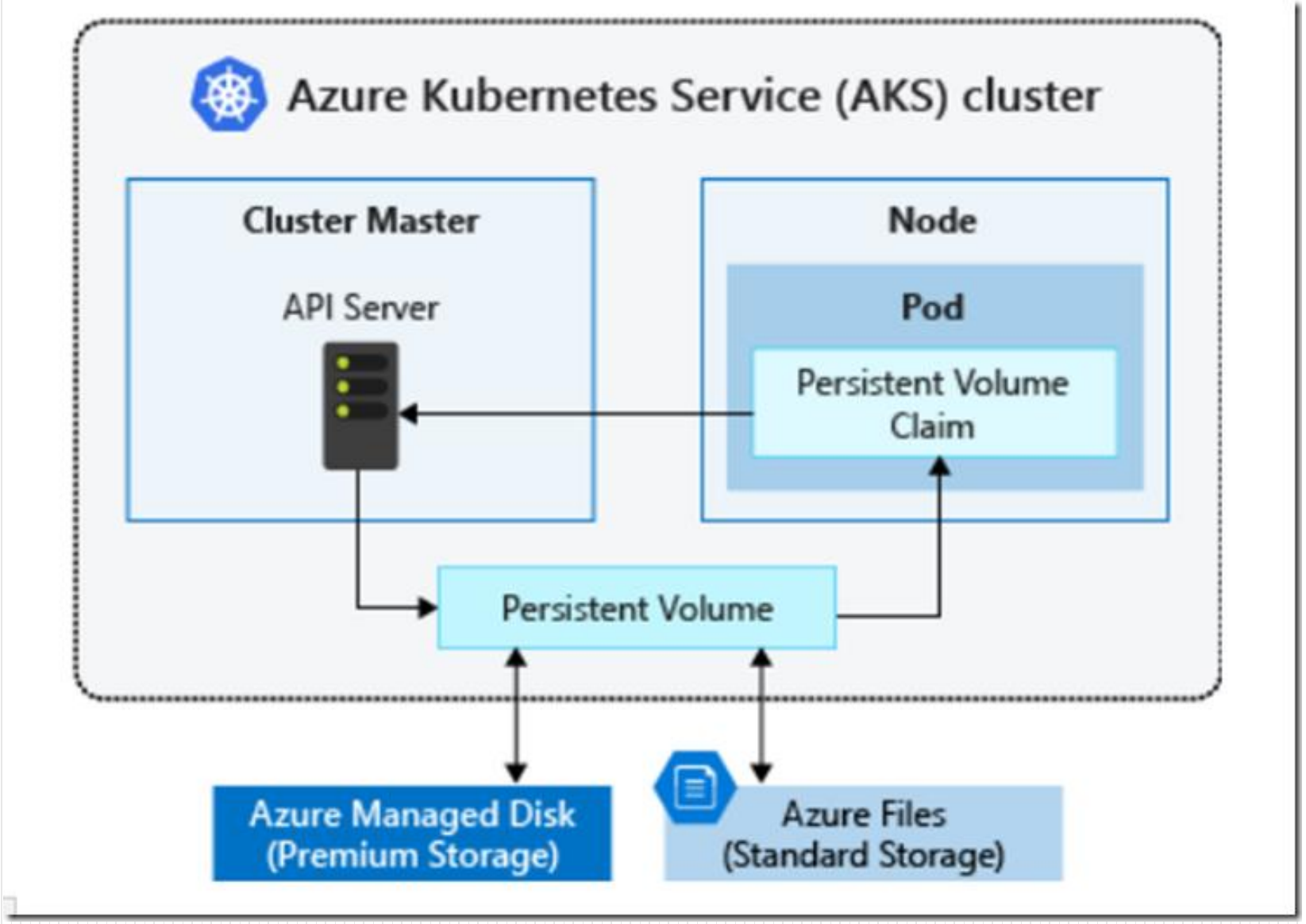
- tmpfs mounts allow for two configuration options, neither of which is required.
- tmpfs-size Size of the tmpfs mount in bytes. Unlimited by default.
- tmpfs-mode File mode of the tmpfs in octal. For instance, 700 or 0770. Defaults to 1777 or world-writable.

```
docker run -d -it --name tmptest \
  --mount type=tmpfs,destination=/app,tmpfs-mode=1770 \
  nginx:latest
```

Kubernetes Application Data

Volumes

Kubernetes Volumes



Kubernetes Volumes

- A Kubernetes volume has an explicit lifetime - the same as the Pod that encloses it.
- Consequently, a volume outlives any Containers that run within the Pod, and data is preserved across Container restarts.
- When a Pod ceases to exist, the volume will cease to exist, too.
- Kubernetes supports many types of volumes, and a Pod can use any number of them simultaneously

Kubernetes Volumes

- Kubernetes supports several types of Volumes:

- awsElasticBlockStore
- azureDisk
- azureFile
- cephfs
- cinder
- csi
- fc (fibre channel)
- glusterfs
- hostPath
- iscsi
- local
- nfs
- persistentVolumeClaim

Kubernetes Volume

- This Pod has a Volume of type emptyDir that lasts for the life of the Pod, even if the Container terminates and restarts.

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-storage
      mountPath: /data/redis
  volumes:
  - name: redis-storage
    emptyDir: {}
```

Persistent Volumes

- A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes.
- It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV.
- This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

Persistent Volume Claims

- A PersistentVolumeClaim (PVC) is a request for storage by a user. It is similar to a Pod.
- Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory).
- Claims can request specific size and access modes (e.g., they can be mounted once read/write or many times read-only).
- While PersistentVolumeClaims allow a user to consume abstract storage resources, it is common that users need PersistentVolumes with varying properties, such as performance, for different problems.

Hostpath Persistent Volume

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

Hostpath Persistent Volume Claim

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
accessModes:
  - ReadWriteOnce
resources:
  requests:
    storage: 3Gi
```

Pod with Persistent Volume

```
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
        claimName: task-pv-claim
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
```

Retain Policy

- Current reclaim policies are:

- Retain – manual reclamation
- Recycle – basic scrub (rm -rf /thevolume/*)
- Delete – associated storage asset such as AWS EBS, GCE PD, Azure Disk, or OpenStack Cinder volume is deleted

- Currently, only NFS and HostPath support recycling. AWS EBS, GCE PD, Azure Disk, and Cinder volumes support deletion.

Access Mode

- The access modes are:

- ReadWriteOnce — the volume can be mounted as read-write by a single node
- ReadOnlyMany — the volume can be mounted read-only by many nodes
- ReadWriteMany — the volume can be mounted as read-write by many nodes

Selector

- Claims can specify a label selector to further filter the set of volumes. Only the volumes whose labels match the selector can be bound to the claim. The selector can consist of two fields:

- matchLabels - the volume must have a label with this value
- matchExpressions - a list of requirements made by specifying key, list of values, and operator that relates the key and values. Valid operators include In, NotIn, Exists, and DoesNotExist.

Persistent Volume Claims

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 8Gi
  storageClassName: slow
  selector:
    matchLabels:
      release: "stable"
    matchExpressions:
      - {key: environment, operator: In,
values: [dev]}
```

NFS Persistent Volume

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
persistentVolumeReclaimPolicy: Recycle
storageClassName: slow
mountOptions:
  - hard
  - nfsvers=4.1
nfs:
  path: /tmp
  server: 172.17.0.2
```

Block Persistent Volume

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: block-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
persistentVolumeReclaimPolicy: Retain
fc:
  targetWWNs: ["50060e801049cfd1"]
  lun: 0
  readOnly: false
```

Block Persistent Volume Claims

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: block-pvc
spec:
  accessModes:
    - ReadWriteOnce
volumeMode: Block
resources:
  requests:
    storage: 10Gi
```

ISCSI Persistent Volume Claims

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: iscsi-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  iscsi:
     targetPortal: 10.16.154.81:3260
     portals: ['10.16.154.82:3260', '10.16.154.83:3260']
     iqn: iqn.2014-12.example.server:storage.target00
     lun: 0
     fsType: 'ext4'
     readOnly: false
     chapAuthDiscovery: true
     chapAuthSession: true
     secretRef:
       name: chap-secret
```

Kubernetes Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-block-volume
spec:
  containers:
    - name: fc-container
      image: fedora:26
      command: ["/bin/sh", "-c"]
      args: [ "tail -f /dev/null" ]
      volumeDevices:
        - name: data
          devicePath: /dev/xvda
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: block-pvc
```

Volume Snapshot

- A VolumeSnapshot represents a snapshot of a volume on a storage system.
- VolumeSnapshotContent and VolumeSnapshot API resources are provided to create volume snapshots for users and administrators.
- A VolumeSnapshotContent is a snapshot taken from a volume in the cluster that has been provisioned by an administrator. It is a resource in the cluster just like a PersistentVolume is a cluster resource.
- VolumeSnapshot is a request for snapshot of a volume by a user. It is similar to a PersistentVolumeClaim.

Volume Snapshot

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: restore-pvc
spec:
  storageClassName: csi-hostpath-sc
  dataSource:
    name: new-snapshot-test
    kind: VolumeSnapshot
    apiGroup: snapshot.storage.k8s.io
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Volume Snapshot Content

```
apiVersion: snapshot.storage.k8s.io/v1beta1
kind: VolumeSnapshotContent
metadata:
  name: snapcontent-72d9a349-aacd-42d2-a240-d775650d2455
spec:
  deletionPolicy: Delete
  driver: hostpath.csi.k8s.io
  source:
    volumeHandle: ee0cfb94-f8d4-11e9-b2d8-0242ac110002
  volumeSnapshotClassName: csi-hostpath-snapclass
  volumeSnapshotRef:
    name: new-snapshot-test
    namespace: default
    uid: 72d9a349-aacd-42d2-a240-d775650d2455
```

CSI Volume Cloning

- The CSI Volume Cloning feature adds support for specifying existing PVC s in the dataSource field to indicate a user would like to clone a Volume .

- A Clone is defined as a duplicate of an existing Kubernetes Volume that can be consumed as any standard Volume would be. The only difference is that upon provisioning, rather than creating a “new” empty Volume, the back end device creates an exact duplicate of the specified Volume.

Storage Class

- A StorageClass provides a way for administrators to describe the “classes” of storage they offer.
- Different classes might map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators.
- Kubernetes itself is unopinionated about what classes represent.
- This concept is sometimes called “profiles” in other storage systems

Volume Binding Mode

- By default, the Immediate mode indicates that volume binding and dynamic provisioning occurs once the PersistentVolumeClaim is created.
- A cluster administrator can address this issue by specifying the WaitForFirstConsumer mode which will delay the binding and provisioning of a PersistentVolume until a Pod using the PersistentVolumeClaim is created.

Storage Class

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
reclaimPolicy: Retain
allowVolumeExpansion: true
mountOptions:
  - debug
volumeBindingMode: Immediate
```

Dynamic Provisioning

- Dynamic volume provisioning allows storage volumes to be created on-demand.
- Without dynamic provisioning, cluster administrators have to manually make calls to their cloud or storage provider to create new storage volumes, and then create PersistentVolume objects to represent them in Kubernetes.
- The dynamic provisioning feature eliminates the need for cluster administrators to pre-provision storage. Instead, it automatically provisions storage when it is requested by users.

Dynamic Provisioning

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

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: claim1
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: fast
  resources:
    requests:
      storage: 30Gi
```

Information Security

Secrets Management

https://kubernetes-security.info/#securing-your-container-images

Overview of Secrets

- To use a Secret, a Pod needs to reference the Secret. A Secret can be used with a Pod in three ways:

- As files in a volume mounted on one or more of its containers.
- As container environment variable.
- By the kubelet when pulling images for the Pod.

Types of Secret

- When creating a Secret, you can specify its type using the type field of a Secret resource, or certain equivalent kubectl command line flags (if available).
- The type of a Secret is used to facilitate programmatic handling of different kinds of confidential data.
- Kubernetes provides several builtin types for some common usage scenarios. These types vary in terms of the validations performed and the constraints Kubernetes imposes on them

Builtin Type	Usage
`Opaque`	arbitrary user-defined data
`kubernetes.io/service-account-token`	service account token
`kubernetes.io/dockercfg`	serialized `~/.dockercfg` file
`kubernetes.io/dockerconfigjson`	serialized `~/.docker/config.json` file
`kubernetes.io/basic-auth`	credentials for basic authentication
`kubernetes.io/ssh-auth`	credentials for SSH authentication
`kubernetes.io/tls`	data for a TLS client or server
`bootstrap.kubernetes.io/token`	bootstrap token data

Service account token Secrets

- apiVersion: v1
- kind: Secret
- metadata:
- name: secret-sa-sample
- annotations:
- kubernetes.io/service-account.name: "sa-name"
- type: kubernetes.io/service-account-token
- data:
- # You can include additional key value pairs as you do with Opaque Secrets
- extra: YmFyCg==

Docker config Secrets

- You can use one of the following type values to create a Secret to store the credentials for accessing a Docker registry for images.
- kubernetes.io/dockercfg
- kubernetes.io/dockerconfigjson

- The kubernetes.io/dockercfg type is reserved to store a serialized ~/.dockercfg which is the legacy format for configuring Docker command line. When using this Secret type, you have to ensure the Secret data field contains a .dockercfg key whose value is content of a ~/.dockercfg file encoded in the base64 format.
- The kubernetes.io/dockerconfigjson type is designed for storing a serialized JSON that follows the same format rules as the ~/.docker/config.json file which is a new format for ~/.dockercfg. When using this Secret type, the data field of the Secret object must contain a .dockerconfigjson key, in which the content for the ~/.docker/config.json file is provided as a base64 encoded string.

Docker config Secrets

- Below is an example for a kubernetes.io/dockercfg type of Secret:

- apiVersion: v1
- kind: Secret
- metadata:
- name: secret-dockercfg
- type: kubernetes.io/dockercfg
- data:
- .dockercfg: |
- "<base64 encoded ~/.dockercfg file>"

Basic authentication Secret

- The kubernetes.io/basic-auth type is provided for storing credentials needed for basic authentication. When using this Secret type, the data field of the Secret must contain one of the following two keys:

- username: the user name for authentication;
- password: the password or token for authentication.

- Both values for the above two keys are base64 encoded strings. You can, of course, provide the clear text content using the stringData for Secret creation.

SSH authentication secrets

- The builtin type kubernetes.io/ssh-auth is provided for storing data used in SSH authentication. When using this Secret type, you will have to specify a ssh-privatekey key-value pair in the data (or stringData) field as the SSH credential to use.

- apiVersion: v1
- kind: Secret
- metadata:
- name: secret-ssh-auth
- type: kubernetes.io/ssh-auth
- data:
- # the data is abbreviated in this example
- ssh-privatekey: |
- MIIEpQIBAAKCAQEAulqb/Y ...

TLS secrets

- Kubernetes provides a builtin Secret type kubernetes.io/tls for storing a certificate and its associated key that are typically used for TLS .
- This data is primarily used with TLS termination of the Ingress resource, but may be used with other resources or directly by a workload.
- When using this type of Secret, the tls.key and the tls.crt key must be provided in the data (or stringData) field of the Secret configuration, although the API server doesn't actually validate the values for each key.

TLS secrets

- apiVersion: v1
- kind: Secret
- metadata:
- name: secret-tls
- type: kubernetes.io/tls
- data:
- # the data is abbreviated in this example
- tls.crt: |
- MIIC2DCCAcCgAwIBAgIBATANBgkqh ...
- tls.key: |
- MIIEpgIBAAKCAQEA7yn3bRHQ5FHMQ ...

Using Secrets as files from a Pod

- To consume a Secret in a volume in a Pod:

- Create a secret or use an existing one. Multiple Pods can reference the same secret.
- Modify your Pod definition to add a volume under .spec.volumes[]. Name the volume anything, and have a .spec.volumes[].secret.secretName field equal to the name of the Secret object.
- Add a .spec.containers[].volumeMounts[] to each container that needs the secret. Specify .spec.containers[].volumeMounts[].readOnly = true and .spec.containers[].volumeMounts[].mountPath to an unused directory name where you would like the secrets to appear.
- Modify your image or command line so that the program looks for files in that directory. Each key in the secret data map becomes the filename under mountPath.

Projection of Secret keys to specific paths

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      items:
      - key: username
        path: my-group/my-username
```

Projection of Secret keys to specific paths

- What will happen:

- username secret is stored under /etc/foo/my-group/my-username file instead of /etc/foo/username.
- password secret is not projected.

- If .spec.volumes[].secret.items is used, only keys specified in items are projected. To consume all keys from the secret, all of them must be listed in the items field. All listed keys must exist in the corresponding secret. Otherwise, the volume is not created.

Using Secrets as environment variables

- To use a secret in an environment variable in a Pod:

- Create a secret or use an existing one. Multiple Pods can reference the same secret.
- Modify your Pod definition in each container that you wish to consume the value of a secret key to add an environment variable for each secret key you wish to consume. The environment variable that consumes the secret key should populate the secret's name and key in env[].valueFrom.secretKeyRef.
- Modify your image and/or command line so that the program looks for values in the specified environment variables.

Using Secrets as environment variables

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: redis
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: username
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: password
  restartPolicy: Never
```

Immutable Secrets

- FEATURE STATE: Kubernetes v1.21 [stable] : **immutable: true**

- The Kubernetes feature Immutable Secrets and ConfigMaps provides an option to set individual Secrets and ConfigMaps as immutable. For clusters that extensively use Secrets (at least tens of thousands of unique Secret to Pod mounts), preventing changes to their data has the following advantages:

- protects you from accidental (or unwanted) updates that could cause applications outages
- improves performance of your cluster by significantly reducing load on kube-apiserver, by closing watches for secrets marked as immutable.

Questions?

andry.cheredarchuk@gmail.com