Persistent Container Storage

WRITTEN BY ALAN HOHN, LOCKHEED MARTIN FELLOW

CONTENTS

OVERVIEW

Ephemeral storage is a major selling point of containers. "Start a container from an image. Make whatever changes you want. Then you stop it and start a new one. Look, a whole new file system that resets back to the content of the image!"

In Docker terms, that might look like this:

```
# docker run -it centos
[root@d42876f95c6a /]# echo "Hello world" > /hello-file
[root@d42876f95c6a /]# exit
exit
# docker run -it centos
[root@a0a93816fcfe /]# cat /hello-file
cat: /hello-file: No such file or directory
```

When we build applications around containers, this ephemeral storage is incredibly useful. It makes it easy to scale horizontally: we just create multiple instances of containers from the same image, and each one gets its own isolated file system. It makes it easy to upgrade: we just create a new version of the image, and we don't have to worry about upgrade-in-place or capturing anything from existing container instances. It makes it easy to move from a single system to a cluster, or from on-premises to cloud: we only need to make sure the cluster or cloud can access our image in a registry. And it makes it easy to recover: no matter what our application might have done to its file system on its way to a horrible crash, we just start a new, fresh container instance from the image and it's like the failure never happened.

So, we don't want our container engine to stop providing ephemeral, temporary storage. But we do have a problem when we transition from tutorial examples to real applications. Real applications must keep state somewhere. Often, we push our state back into some data store (SQL-based or NoSQL-based). But that just raises the question of where to put the data store application.

Is it also in a container? Ideally, the answer is "yes," so we can take advantage of the same rolling upgrades, redundancy, and failover that we use for our application layer. To run our data store in a container, however, we can no longer be satisfied with just ephemeral, temporary storage. Our container instances need to be able to access persistent storage.

For simple cases where we just run our Docker containers directly, this is easy. We have two main choices: we can identify a directory on the host file system, or we can have Docker manage the storage for us. Here's how it looks when Docker manages the storage:

```
# docker volume create data
data
# docker run -it -v data:/data centos
[root@5238393087ae /]# echo "Hello world" > /data/
hello-file
[root@5238393087ae /]# exit
exit
# docker run -it -v data:/data centos
[root@e62608823cd0 /]# cat /data/hello-file
Hello world
```

Docker does not keep the root file system from the first container, but it does keep the "data" volume, and that same volume is mounted in the second container as well, so the storage is persistent.

This works on a single system, but access to persistent storage gets more complicated in a clustered container environment like Kubernetes or Docker Swarm. If our data store container might get started on any one of hundreds of nodes and might migrate from one node to another at any time, we can't just rely on one server's file system to store the data. We need a storage solution that is aware of containers and distributed processing and can seamlessly integrate.

This Refcard will describe the solution to this need for container-aware storage and will show how getting the storage solution right is a key element of building reliable containerized applications that excel in production.

CONTAINER STORAGE REQUIREMENTS

Before looking at solutions for container storage, we should look at what we want the solution to look like, so we'll better understand the design decisions for the container storage solutions that are out there.

REDUNDANT

One of the reasons for moving our application into containers and deploying those containers into an orchestration environment is that we can have many physical nodes and can tolerate the failure of some of those nodes. In just the same way, we want our storage solution to be able to tolerate disk and node failure and keep our application running. With storage, the need for redundancy is even more important because we can't afford to lose any data even if we have some downtime.

DISTRIBUTED

The need for redundant storage drives us to some kind of distributed solution, at least with respect to disks. But we also want distributed storage for performance. As we scale our container environment up to hundreds or thousands of nodes, we don't want those nodes to be competing for data on the same few disks. Also, as we expand our environment to multiple geographic regions to reduce latency for our users, we also want to distribute our storage geographically so access to storage is fast from anywhere.

DYNAMIC

Container architectures are undergoing continuous change. New versions are built, updates are rolled in incrementally, applications are being added and removed. Test instances are created, put through automated tests, and destroyed. In this architecture, it must be possible to provision and release storage dynamically as well. In fact, provisioning storage should be declarative in the same way that we can declare container instances, services, and network connectivity.

FLEXIBLE

Container technology is moving quickly, and we need to be able to introduce new storage solutions or port our application to new environments with different underlying storage infrastructure. Our storage solution needs to be able to support any underlying infrastructure, from a single machine used by a developer for testing purposes, to an on-premise environment, to a public cloud deployment.

TRANSPARENT

We need to provide storage to any kind of application, and we need to update our storage solution over time. This means we can't tie our application to a just one storage solution. Instead, storage needs to look native, whether that means looking like a file system, or looking like some existing, understood API.

CLOUD NATIVE STORAGE

Another way to put it is that we want our container storage solution to be "Cloud Native." The Cloud Native Computing Foundation (CNCF) has identified three properties for cloud native systems. We can apply these to storage:

a. **Container packaged**. Ultimately, our physical or virtual disks exist outside the container, but we want to present storage specifically to containers (so that containers are not sharing storage unless that was specifically requested). Additionally, we may want to containerize the storage control software itself, so we can use the advantages of containerization to manage and update the software that manages storage.

b. **Dynamically managed**. For continuous deployment of stateful containers, we need to be able to allocate storage for new containers and clean up storage that is no longer needed, without manual intervention by some administrator.

c. **Microservices oriented**. When we define a container, it should explicitly express its dependency on storage. Additionally, the storage control software itself should be based on microservices so it's easier to scale and to distribute geographically.

The CNCF Storage Working Group is working on a whitepaper covering the CNCF storage landscape. In the meantime, there are some good resources, including a primer on cloud native storage and 8 principles for cloud native storage.

PROVISIONING CONTAINER STORAGE

To answer this container storage need, both Kubernetes

and Docker Swarm provide a set of declarative resources for provisioning and attaching storage to containers. These storage capabilities are built on top of some storage infrastructure. Later in this Refcard we'll look at some choices for container storage, but first let's look at how each of these two environments allows containers to declare storage dependencies.

KUBERNETES

In Kubernetes, containers live in Pods. Each pod includes one or more containers that all share the same network stack and storage. Storage is defined in the volumes section of the pod definition, and volumes are available to be mounted in any container in the pod.

For example, here is a Kubernetes pod definition using an emptyDir volume to share information between two containers. As the name suggests, the emptyDir volume starts out empty, but it stays persistent while the pod is allocated to a node (which means it survives ordinary container crashes but doesn't survive node failure or pod deletion).

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-storage
spec:
  restartPolicy: Never
  volumes:
  - name: shared-data
    emptyDir: {}
  containers:
  - name: nginx-container
    image: nginx
    volumeMounts:
    - name: shared-data
      mountPath: /usr/share/nginx/html
  - name: debian-container
    image: debian
    volumeMounts:
    - name: shared-data
      mountPath: /pod-data
    command: [“/bin/sh”]
    args: [“-c”, “echo Hello from the debian container
> /pod-data/index.html”]
```

If we save this to a file called `two-containers.yaml` and deploy it to Kubernetes using `kubectl create -f two-containers.yaml`, we can browse to the NGINX server using the pod's IP address and retrieve the created `index.html` file.

This is an important example, because it shows how Kubernetes allows us to declare a storage dependency in a pod using the volumes section. However, this still isn't true permanent storage.

If our Kubernetes container is using Amazon Web Services Elastic Compute Cloud (AWS EC2), an example with permanent storage might look like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: webserver
spec:
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: web-files
  volumes:
  - name: web-files
    awsElasticBlockStore:
      volumeID: <volume-id>
      fsType: ext4
```

For this example, we can destroy and create the pod again, and the same storage will be provided to the new pod, no matter what node is used to run the container. However, this example still does not provide dynamic storage, as we must separately create the Elastic Block Store (EBS) volume before we can create our pod.

To get dynamic storage from Kubernetes, we need two other important concepts. The first is storageClass. Kubernetes allows us to create a storageClass resource that collects information about a storage provider. We then combine this with a persistentVolumeClaim, a resource that allows us to request storage from a storageClass dynamically, with Kubernetes requesting storage for us from the storageClass we've chosen. Here's an example, still using AWS EBS:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: file-store
provisioner: kubernetes.io/aws-ebs
parameters:
  type: io1
  zones: us-east-1d, us-east-1c
  iopsPerGB: “10”
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: web-static-files
spec:
  resources:
    requests:
      storage: 8Gi
  storageClassName: file-store
---
```

CODE CONTINUED ON NEXT PAGE

```
apiVersion: v1
kind: Pod
metadata:
  name: webserver
spec:
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: web-files
  volumes:
  - name: web-files
    persistentVolumeClaim:
      claimName: web-static-files
```

As you can see, we still use the volumes section of the pod to specify our need for storage, but we use a separate PersistentVolumeClaim to ask Kubernetes to provision the resource for us. In general, the cluster administrator would deploy StorageClasses **once per cluster** to represent the available underlying storage. Then the application developer would specify PersistentVolumeClaims once per application when storage is first needed. The pod is then deployed and replaced as needed for application upgrades without losing the data in storage.

DOCKER SWARM

Docker Swarm leverages the same core volume management capabilities we saw with a single-node Docker volume, but with the ability to provide storage to a container on any node. To provision containers in Docker Swarm, we use the docker stack command together with a Docker Compose file. For example:

```
version: "3"
services:
  webserver:
    image: nginx
    volumes:
      - web-files:/usr/share/nginx/html
volumes:
  web-files:
    driver: storageos
    driver-opts:
      size: 20
      storageos.feature.replicas: 2
```

When we use docker stack deploy, Docker Swarm will create the web-files volume if it doesn't exist. This volume will be retained even if we remove the stack with docker stack rm.

Overall, we can see how both Kubernetes and Docker Swarm meet our criteria for cloud native storage. They allow containers to declare storage as a dependency, and they dynamically manage storage to make it available to an application on-demand. They are also able to provide this storage to a container no matter where in the cluster the container is running.

Of course, to provide this dynamic, distributed storage, both Kubernetes and Docker Swarm rely on configuring some underlying storage infrastructure. Let's now look at our options and how we can decide what kind of storage infrastructure we want for our container environment.

OPTIONS FOR CONTAINER STORAGE INFRASTRUCTURE

There are numerous storage options out there for both Kubernetes and Docker Swarm, but we can group them into a few categories. For each category, we'll look at options within and outside the cloud and discuss how well it meets our overall requirements toward cloud native storage.

CATEGORY	OUTSIDE THE CLOUD	INSIDE THE CLOUD
Raw Block Device	• Simplest option. • High performing. • Allows container direct access to disk. • Ties container to node or disk interface (e.g. SCSI, Fibre Channel).	• Can use underlying cloud resources (e.g. EBS, Azure, OpenStack Cinder). • Extra work to make storage usable (e.g. partition, format).
Network Attached Storage	• Uses well understood protocols such as NFS and iSCSI. • Can integrate with existing on-premise NAS. • Lack of data locality can hurt performance.	• Can leverage in-cloud storage providers (Managed NFS). • Storage is easy to access from inside and outside containers. • Not optimized for the dynamic addition and removal of container volumes.
Distributed File Systems	• Can operate on same container infrastructure. • Opportunity for excellent data locality. • Storage controller software can run within container environments.	• Can operate on same container infrastructure. • Opportunity for excellent data locality, even in geo-distributed situations. • Requires some additional provisioning for storage and controller software.

STORAGEOS

Cloud Native Persistent Storage for Containers

Containers have made app deployment lightweight and portable. Your data should be too. Yet traditional storage solutions don't support today's modern development.

StorageOS delivers a cloud native storage solution to run stateful applications in containers.

GET THE STORAGEOS PLATFORM ARCHITECTURE OVERVIEW

StorageOS is a cloud native, persistent storage solution for containers that makes it easy to build stateful containerized apps. It supports production workloads for transactional databases, messaging systems and other business-critical data stores. StorageOS enhances your applications adding: high availability, rapid recovery, data encryption and the lowest possible latency. Benefit from container storage, as agile as your application, with automated policy management on par with traditional enterprise storage solutions.

Built to run with any stateful application, on any infrastructure with any orchestrator and as a container, StorageOS easily integrates with your favorite platforms – Kubernetes, OpenShift or Docker.

Object Stores	• Suitable for file transfer but not read-write random access. • Typically requires additional underlying storage infrastructure.	• Can use underlying cloud resources (e.g. S3, OpenStack Swift). • Suitable for file transfer but not read-write random access.
Software Defined Storage	• Able to specify and receive performance guarantees. • Can insert value-added services such as data de-duplication and snapshots.	• Able to specify and receive performance guarantees. • Aligns with cloud native storage. • Can insert value-added services such as data de-duplication and snapshots.

The final category, Software Defined Storage, is not a new concept but is becoming a more popular term. It continues the trend toward storage abstraction that started with logical storage in the Redundant Array of Independent Disks (RAID) and Logical Volume Manager (LVM), then was extended with virtualized storage through Storage Area Network (SAN) and distributed file systems such as Ceph and Gluster, but it adds a storage abstraction layer that can incorporate de-duplication, built-in backup and archiving, change auditing to establish data provenance, and snapshot capabilities. For persistent container storage, Software Defined Storage operates in a very similar way to distributed file systems, with a single API to provision and manage storage and the ability to localize data at the point of use, but it includes other capabilities that may be desirable from a broader storage management standpoint.

While there's no one "right answer" to the type of persistent storage we should deploy to our container environment, it is important to refer to the list of required elements to make sure we wind up with declarative performant storage, even as our container environment scales to multiple geographic regions and a large number of nodes.

APPLICATION EXAMPLE

To complete our look at persistent container storage, let's deploy a Spring Boot application that uses PostgreSQL. We'll focus on Kubernetes for this example, but the same ideas apply in Docker Swarm.

SECRETS AND CONFIGURATION

So far, our persistent storage discussion has been about files and volumes, and for good reason, since most applications see storage in those terms. However, sometimes we need to provide small pieces of information to our containerized applications, including configuration files, database credentials, and environment variables. For these cases, we want the ability to maintain this information securely and keep it during application rollover or updates, but we don't want to bundle it with the application because it might be specific to the environment or information that must be kept secret.

For small files and variables, it seems like a waste to provide a whole storage volume (in addition to making it more complicated to update from outside the container). Instead, both Kubernetes and Docker Swarm have explicit support for storing configuration data and secrets and providing them to containers.

Our Spring Boot application needs database connection information for the PostgreSQL database. Knowing that we need different information in development and production, we will use variables in `application.properties`:

```
spring.datasource.url=${DB_URL}
spring.datasource.username=${DB_USER}
spring.datasource.password=${DB_PASS}
spring.datasource.driver-class-name=${DB_DRIVER}
```

To provide this information to our container, we will first declare a ConfigMap and a Secret.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: myapp-config
  namespace: myapp
data:
  database.url: jdbc:postgresql://mydb.myapp.pod/myapp
  database.username: dbuser
  database.driver: org.postgresql.Driver
```

There are a couple things to note here. First, we use a namespace to keep the resources for "myapp" separate from other applications. Second, we assume we have deployed Kubernetes DNS to detect our PostgreSQL Pod and make a DNS entry for it. Finally, note that we do not include the password, because we need to use a *Secret* so it is held in an encrypted form.

For our secret, we'll use the Kubernetes command line because we just have a single value to store:

```
# kubectl create secret generic myapp-secret
--namespace=myapp \
  --from-literal=password='correcthorsebatterystaple'
```

APPLICATION DEPLOYMENT

We can now use our ConfigMap and Secret in the Pod definition for our application:

```
kind: Pod
apiVersion: v1
metadata:
  name: webapp
  namespace: myapp
spec:
  containers:
  - image: registry.mycompany.com/myapp
    name: myapp
    env:

    - name: DB_URL
      valueFrom:
        configMapKeyRef:
          name: myapp-config
          key: database.url
    - name: DB_USER
      valueFrom:
        configMapKeyRef:
          name: myapp-config
          key: database.username
    - name: DB_DRIVER
      valueFrom:
        configMapKeyRef:
          name: myapp-config
          key: database.driver
    - name: DB_PASS
      valueFrom:
        secretKeyRef:
          name: myapp-secret
          key: password
```

While this example uses environment variables, ConfigMaps and Secrets can be treated as volumes in Kubernetes, so we could also include the application.properties file or a Spring Boot YAML configuration file in a Secret, and then have Kubernetes place it in the file system of our container so our application could load it. This would allow us to avoid editing the Pod definition to add new properties.

POSTGRESQL DATABASE

As the last step in our application example, let's combine secrets and volumes to show how we might provide persistent storage for the PostgreSQL database that supports our Spring Boot application.

We are going to provide our PostgreSQL container with two separate persistent volumes. The first will be used for the PostgreSQL data directory, and the second for backups and a write-ahead log (WAL). The WAL volume would allow us to configure a PostgreSQL standby server, though actually configuring PostgreSQL in this active/passive failover configuration is outside the scope of this Refcard.

We'll provide storage to our database using StorageOS, a Software Defined Storage solution. Our Kubernetes cluster will use the StorageOS API to provision the requested storage. We start by creating a Secret to hold the information needed to connect to the StorageOS API. This secret has three values, so we'll use a YAML definition. To do this, Kubernetes requires us to base-64 encode each value. Then we can declare the Secret:

```
kind: Secret
apiVersion: v1
metadata:
  namespace: storageos
  name: storageos-api

type: "kubernetes.io/storageos"
data:
  apiAddress: <base 64 encoded URL>
  apiUsername: <base 64 encoded username>
  apiPassword: <base 64 encoded password>
```

Next, we configure the storageClass using that secret:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
metadata:
  name: fast
provisioner: kubernetes.io/storageos
parameters:
  pool: default
  fsType: ext4
  adminSecretNamespace: storageos
  adminSecretName: storageos-api
```

Note that the storageClass is kept in the default namespace since we'll use it with many applications. Finally, we're ready to create our two PersistentVolumeClaims and our database Pod:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pgsql-data
  namespace: default
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: fast
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pgsql-backup
  namespace: default
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 800Gi
  storageClassName: fast
---
kind: Pod
apiVersion: v1
metadata:
```

CODE CONTINUED ON NEXT PAGE

```
  name: mydb
  namespace: default
spec:
  containers:
  - image: postgres:9.4
    name: mydb
    volumeMounts:
    - mountPath: /var/lib/pgsql/data
      name: data
    - mountPath: /backup
      name: backup

  volumes:
  - name: data
    persistentVolumeClaim:
      claimName: pgsql-data
  - name: backup
    persistentVolumeClaim:
      claimName: pgsql-backup
```

We allocate 10GB for pgsql-data. At the moment (Kubernetes 1.9), the ability to expand a persistent volume claim is in alpha and only supported for a few storage classes, so this needs to be large enough to hold the full/expected size of our database. Also, note that while we use a single storage class, it might be beneficial to provision multiple storage classes with different policies to have a cheaper storage option for large volumes, such as backups, where speed is not as critical.

CONCLUSION

In this Refcard, we've looked at the need to provision persistent storage for our containers so we can deploy the stateful parts of our application to our container environment. While the use of a distributed container environment like Kubernetes or Docker Swarm made this more complex, it also created the opportunity for distributed storage, data locality, redundancy, and the ability to deploy our storage controller components directly on the container environment.

In choosing persistent storage infrastructure, we have options that range from basic raw block devices, where we have a simple implementation but limited scalability and redundancy, to sophisticated Software Defined Storage solutions. Software Defined Storage solutions not only guarantee performance, but also allow workloads to be deployed in a platform agnostic manner so the same solution is used for both on-premises and cloud environments. However, whatever solution we choose, we can arrive at a storage solution that our containers can provision dynamically using declarative logic and standard APIs. This gives us the ability to provide our containerized applications with the storage they need while keeping them independent of the underlying infrastructure, which allows us to deploy a broader set of use cases and more complex workloads to cloud native and containerized platforms.

Written by **Alan Hohn**, Lockheed Martin Fellow

Alan Hohn is a Lockheed Martin Fellow with a background as a software architect, lead, and manager. He has a lot of experience re-architecting embedded systems, mostly using distributed Java, and in combining embedded and enterprise approaches. Lately he's been doing quite a bit of work with virtualization, DevOps, and cloud technologies.

DZone, Inc.
150 Preston Executive Dr. Cary, NC 27513
888.678.0399 919.678.0300