YARN-8811: Support Container Storage Interface(CSI) in YARN

Author: Weiwei Yang, Sunil Govindan

Version	Date	Changes
v1	Sep 19, 2018	Initial version
v2	Sep 28, 2018	1) Added more info about source/destination mount points and mount propagation in section 5.2 2) Added user credential handling in section 6 3) Update volume resource definition
v3	Oct 28, 2018	Add controller discovery mechanism
v4	Nov 7, 2018	Add sample validate volume capability workflow chart

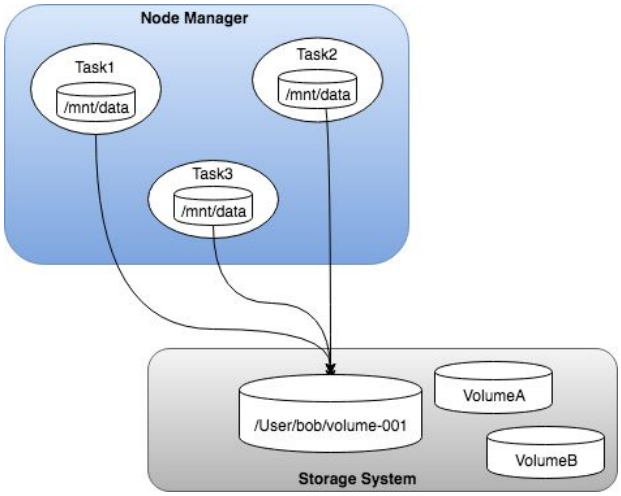
Table 1: Change Log

1. Introduction

The Container Storage Interface (CSI) is a vendor neutral interface to bridge Container Orchestrators (CO) and Storage Providers (SP). COs who adopt CSI can leverage any storage provider, cloud or otherwise. Likewise, CSI enables the SPs to provide storage services to any CO that implements the specification.

Once YARN supports CSI, it will be simple to access various of storage systems in YARN docker containers as long as they are compatible with CSI. With this feature, it enables YARN to fit following scenarios:

- A general solution to tie to external storage systems
- Pre-populated some data in persistent storage for durable access
- Dynamically provision persistent volumes for containers and maintain task states in the storage, so the state can be retained upon restart



CSI defines the protocol of 3 gRPC services: controller, node and identity. SP just needs to implement a driver binary according to these protocols and then COs are able to consume these APIs to access SP's storage services. The detail of the specification can be found here. From high level,

- **Controller**: responsible of controlling and managing the volumes, such as: create, delete, attach/detach, snapshot, etc. This plugin can run anywhere but just single copy.
- **Node**: it formats and mount volumes to the underneath storage. This plugin needs to run on the node where the volume will be provisioned.
- **Identity**: allows a CO to query a plugin for capacities, health, and other metadata. This plugin needs to run with both controller and node plugins.

In this proposal, we introduce a solution for YARN to incorporate these services in response to those requests that ask for persisted volumes. We propose to make this two phase efforts, phase 1 focus on implementing YARN internal modules and protocols to work with CSI, only support pre-provisioned volume in order to simplify the volume lifecycle management; phase 2 we extend to support dynamical provisioned volumes and add security model.

2. Phase-1 Targets

Phase-1 targets are outlined as below

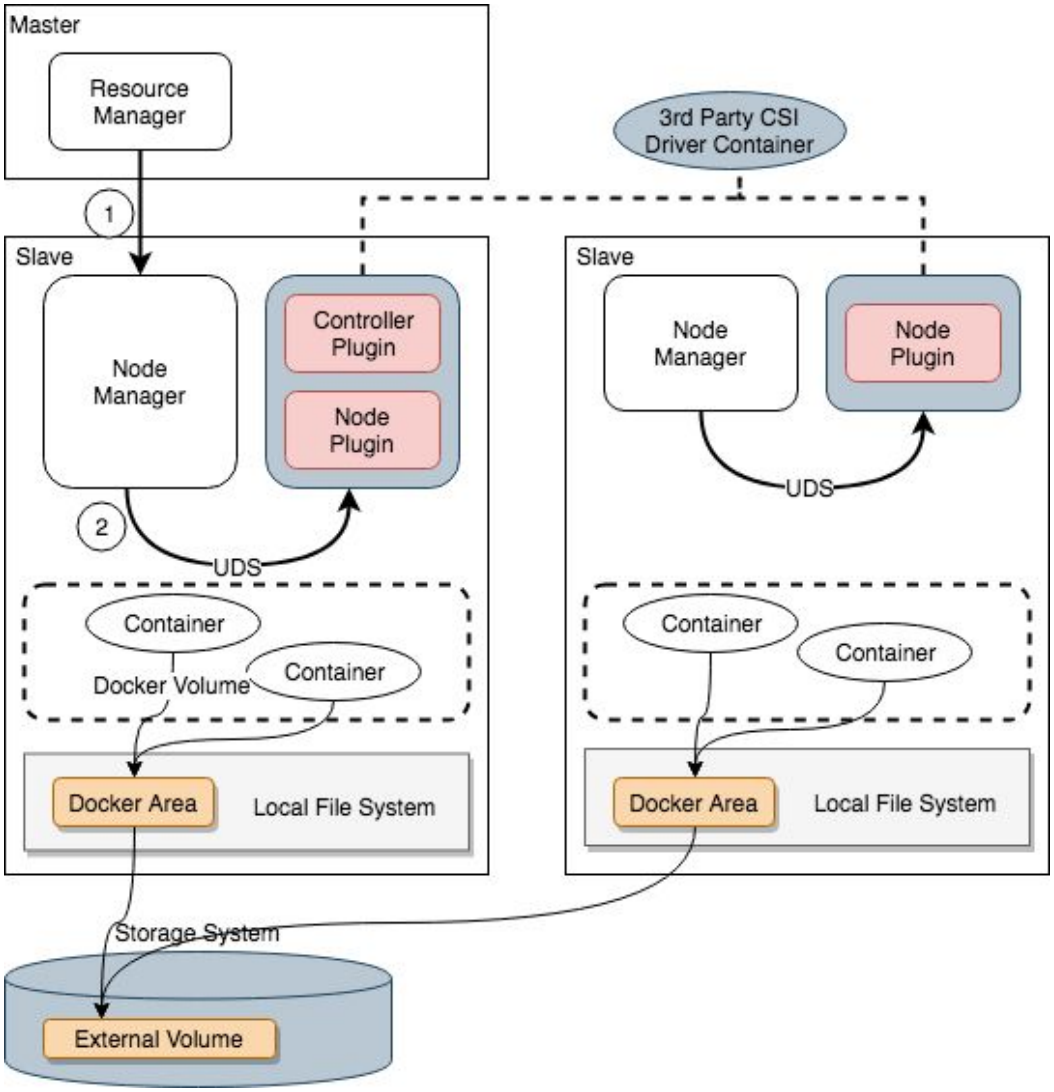
1. Define and implement YARN user level API for CSI volumes
2. Framework support for YARN to interact with CSI driver
3. Support to bind pre-provisioned CSI volumes[1] to docker containers
4. Provide a end-to-end native service example with volume attached

In phase 2, we will further explore how to support dynamical provisioned volumes and how to enhance the security model with external storage systems.

3. Proposal Overview

With CSI abstraction, we are able to add a clean interface in YARN to support external volumes, YARN exposes CSI volume in its public API for user to request (see more in CSI Volume). When process such requests, YARN internally translates them to corresponding CSI API calls, route them to CSI driver and process storage related operations. In YARN, we don't need to maintain any code directly related to a storage vendor, we only communicate with CSI driver using the common interface. An overview of the proposal can be illustrated as below,

[1] There are two volume provisioning pattern in CSI spec, 1) pre-provisioned; 2) dynamically provisioned. See more in *Volume Lifecycle* section of the CSI spec doc.



On every NM, there is one and only one csi-driver-container deployed[2], NM sets up a communication channel with the driver running on the same host, via unix domain socket. In NM process, there will be an CSI driver adaptor delegates all calls between YARN and a CSI driver, including both controller/node/identity API calls. Such adaptor needs to register with RM in order to tell RM which node runs the controller plugin. A typical workflow is,

1) App requests for some containers, and each container needs to mount a directory "/mnt/data" to a pre-provisioned volume "volume-001"
2) RM intercepts the volume info from the AM requests, for each volume request, RM calls the controller plugin (via NM adaptor) to ensure the volume is properly provisioned
3) Scheduler places the containers to certain NMs
4) AM asks NM to launch these containers

[2] How to deploy and manage CSI drivers will be discussed in a separate JIRA. In this proposal, we assume the driver containers are deployed on the cluster and endpoints are available for access.

5) Before NM launch a container, NM calls the node plugin to mount the given volume "volume-001" to a specific mount point on NM's local file system
6) NM launches the container and bind the local mount point to the docker container on "/mnt/data" via docker volume.
7) Container finished execution, NM disables the mount and properly handles the cleanup of the local mounts as well as the volume

4. The Volume Resource

We propose to add Volume as a resource in YARN. A volume represents a storage location with a certain capability that can be accessed by YARN containers, its life cycle is decoupled from container and application. Which means it won't be erased when a container or application is finished (this is very similar to the persistent volumes in k8s). A volume can be mounted to a container on a specific mount path, and containers are accessing the volume via this mount path. Volume resource can be provided by local disk or block device, or from an external storage system.

For the initial version, we limit our scope within:

- Only to support volumes from the storage system that compatible with CSI specification
- Only to support mount volume[3] (a volume mounted with file system and appears as a directory inside the container)
- Only to support volumes that resource is not managed by YARN

4.1 Sample Volume Resource Information

A sample of volume resource information is like below:

```
yarn.io/nfs-volume {
  size : 100,
  unit: Gi,
  ResourceType: countable,
  Tags : ["externalVolume"]
  Attributes : {
    driver: "nfs-csi-driver",
    mountPoint: "/mnt/amt",
    accessMode: "readonly",
```

[3] This is majorly because we concern about the testing effort, the design doesn't limit us only to support mount volumes. It won't have much more work to support block volumes, perhaps no changes at all. How to consume the block device will be handled inside of the app's container.

```
    volumeId: "vol-01" // preprovisioned
  }
}
```

Upon spec defines a volume resource, that connects to an external storage system through "nfs-csi-driver", that mounts a pre-provisioned NFS volume to a local dir "/mnt/amt" inside of the docker container. The spec contains two new concept, "**tags**" and "**attributes**", we'll explain this in 4.2 and 4.3, as an extension to existing resource model.

4.2 Resource Tags

There are 2 types of volumes depending on how resource is managed.

1) Local Volume: resource is provided locally, e.g NM's local disk, device mapper, LVM etc.
2) External Volume: resource is provided by a external storage system, e.g Ozone, EBS

YARN handles them differently. For local volume, RM needs to collect resource capacity from NMs and calculate the capacity like other major resources during the scheduling; for external volume, YARN doesn't manage those resource so it handles them by contacting a CSI driver. To support both scenarios, we propose to add another field in resource information,

- Tags (repeated string)

Tags can be used by YARN to filter resources. Name convention for tags is: **[system:|user:]tagName**. Where *system* and *user* are the two (and only two) supported namespaces for tags, YARN reserves system namespace and user is prohibited to add any system tags. All tags added by user will be auto added into "user" namespace with "user:" prefix. That means user either add tags with name "user:xxx", or without namespace prefix. The tag name cannot contain space, colon, comma or other special chars.

By default, a system tag "**system:default**" is attached to all existing resources, memory/cpu/gpu/fpga. We'll introduce another built-in system tag "**system:externalVolume**" to tag all external volumes. Volumes tagged with "system:externalVolume" will be treated as an unmanaged resource and not be evaluated in the resource calculator, this is used for external volumes; on the other hand, volumes tagged with "system:default" implicates the resource needs to be counted, this is used for local volumes. At this version, we only support volumes with "system:externalVolume" tag.[4]

[4] This is the first step to support resource tagging & filtering. Moving on, we can add user namespace support and let user be able to specify arbitrary tags in the request, YARN is responsible to map requests to resources which has certain tags.

YARN reserves “system:” prefix for internal tags, which means any user specified tag wich this prefix will be treated as invalid. More importantly, system tags are not exposed to users, and there are only “system:default” and “system:externalVolume” tags are supported now.

4.3 Resource Attributes

Currently a resource in YARN consists following information:

- Name (string)
- Units (string)
- ResourceType (COUNTABLE)
- Value (long)
- MinAllocation (long)
- MaxAllocation (long)

This describes a single dimensional resource with single countable value. However for a volume resource, this is not enough, it needs to carry more info such as volume name, access mode, mount path etc. Therefore, we propose to add optional “**Attributes**” field

- Attributes (string-string-map)

The attributes can be used to carry all required info for a resource provider (usually an external resource provider like a storage system) in order to make the resource available for consumption.

4.4 Difference between Tags and Attributes

The difference between attributes and tags is: attributes carries info for a resource provider, those info can be vary between storage vendors; tags carries info for YARN to filter resource when necessary. From user angle, it is allowed that user can add arbitrary attributes as key-value pairs in resource definition, however, we only allow to specify a predefined set of system tags for resources and disallow users to modify them.

5. Communication Channel

5.1 RM and CSI controller plugin

According to the deployment architecture, we need to develop some components in YARN to handle ResourceRequest/SchedulingRequest that carried volume resource, manage the lifecycle of volumes, local mounts, and translate these operations according to CSI protocol.

CSI Volume Processor

An application master processor that implemented for CSI, it intercepts the csi-volume spec from resource/scheduling requests and properly handles the request, then forward the requests to next processor in the chain.

CSI Volume Manager

The CSI volume manager reads the csi-volume spec and handles volume operations by connecting to the controller service on a csi-driver-container. This module manages the lifecycle of CSI volumes in YARN. Note, if there are multiple csi-driver instances running on the cluster, we need same number instance of volume managers to co-work with them.

CSI Driver Client

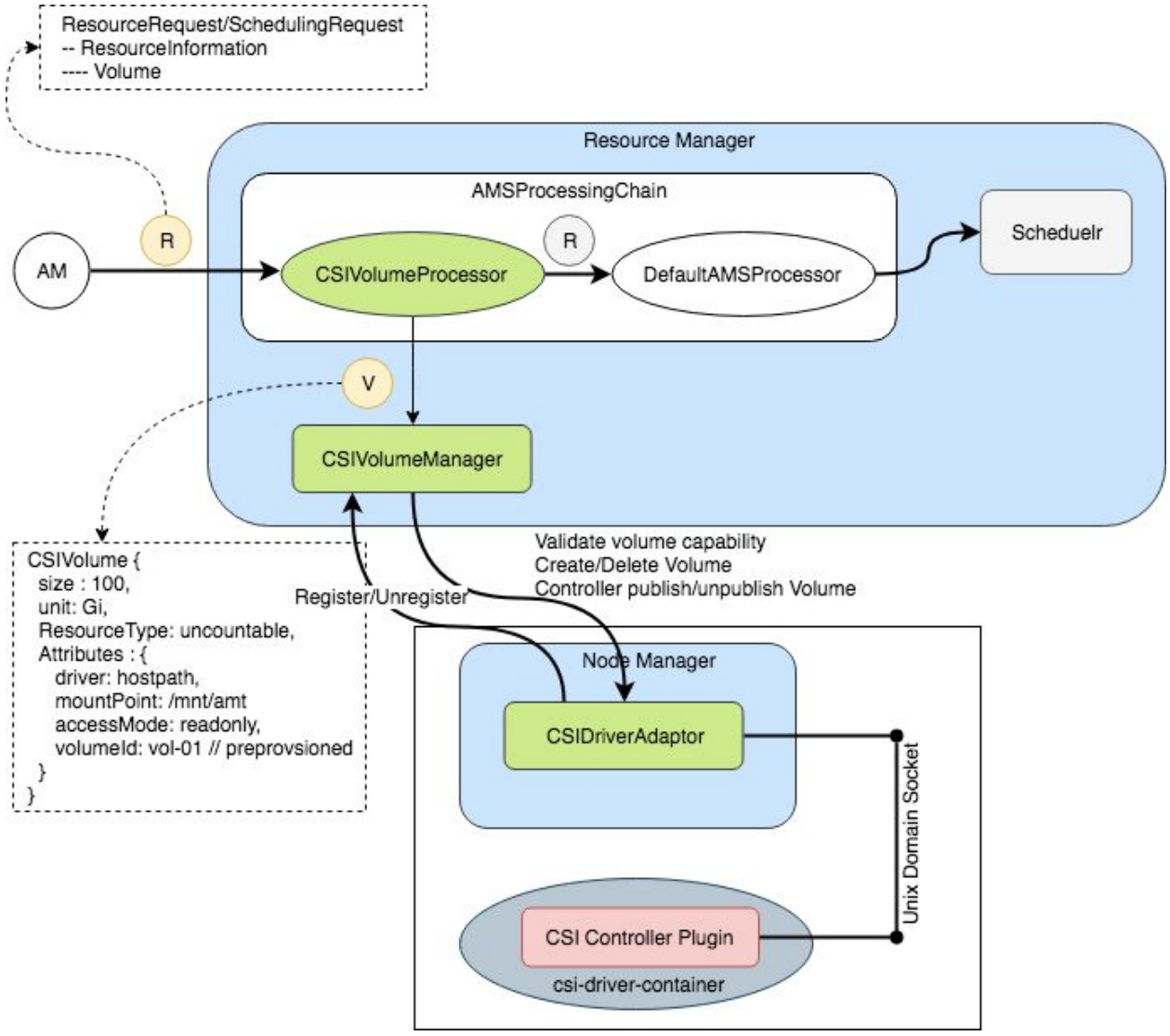
A client library to connect to a certain CSI driver endpoint to handle CSI related operations, this client should be pluggable[5].

CSI Driver Adaptor

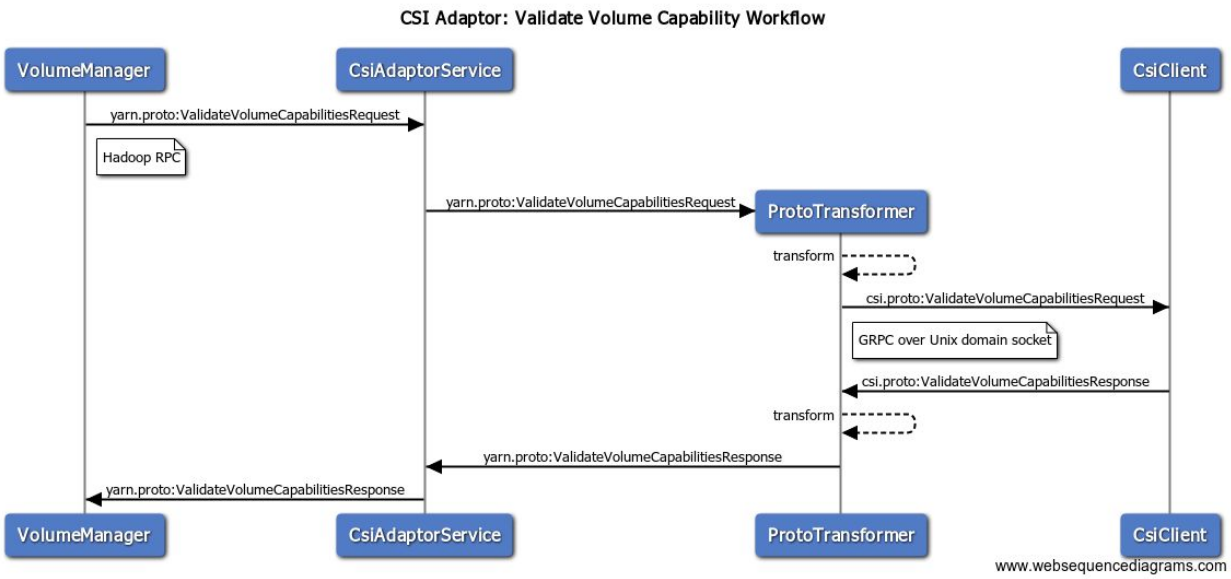
The CSI driver adaptor runs on all node managers, side by side with csi-driver-container. It delegates all invocations between YARN and CSI driver through Unix Domain Socket on the node manager host. The adaptor also needs to register itself to the volume manager and one of them will be the primary talking to the controller plugin.

[5] We can leverage existing CSI client library from gocsi project for testing, but we need to develop our own client library for production.

A more detail view is like following:



The adaptor service runs on NMs so it can hide the CSI protocol messages from YARN components, this helps us to build stable API at YARN side and only need to evolve the adaptor along with CSI spec. An example workflow for validating volume capability with a CSI driver is demonstrated as below.



Controller Service Discovery

In phase 1, we use configuration to identify which node runs the controller plugin, in order to simplify the design. That means if the driver or the node runs controller service collapses, the controller needs to be re-configured to a different node, and the configuration on RM needs to be updated accordingly. Otherwise RM side calls to the driver will all fail.

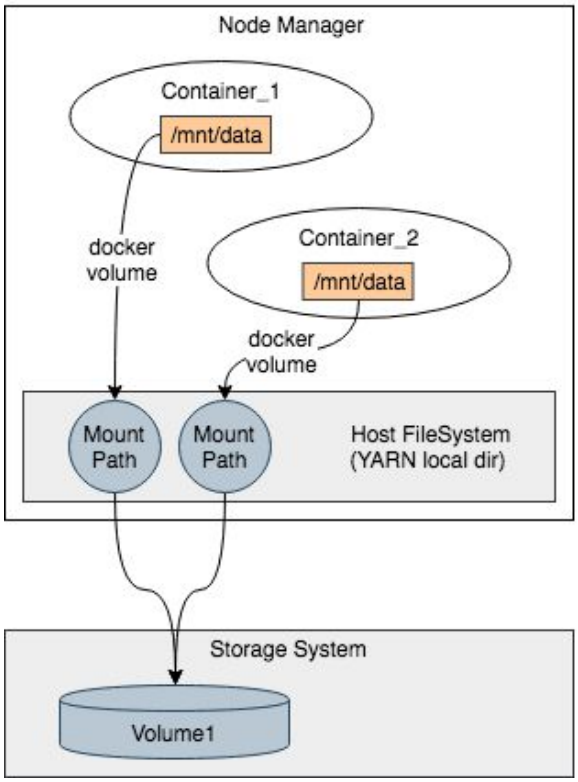
If we fully manage the deployment of the CSI drivers in the future, we can do further improvements such as lead election and failure recovery for the controller plugin. We can add an extra register/sync mechanism between the driver and resource manager in order to improve the usability and fault tolerance.

5.2 NM and CSI node plugin

Before we launch a container, in order to make a volume visible and accessible for a docker container, it must be properly formatted and mounted to a directory. Per CSI spec, this is handled by the node-service of the csi-driver. Related API calls including

- NodeStageVolume
- NodeUnstageVolume
- NodePublishVolume
- NodeUnpublishVolume

When NM starts the container, it firstly extracts the CSI-Volume info from resource, and call the csi-driver on the same node (via UDS) to mount the volume to a *container-csi-mount* directory under YARN local directory, then we mount this *container-csi-mount* to the container in path where user specified in the volume request.




```
Container_1
   yarn.io/csi-volume : {
      value : "10",
      unit : "Gi",
      attributes: {
          volumeName : "test-volume",
          driverName: "hostpath",
          mountPath: "/mnt/data"
      }
   }
```

```
Container_2
   yarn.io/csi-volume : {
      value : "5",
      unit : "Gi",
      attributes: {
          volumeName : "test-volume",
          driverName: "hostpath",
          mountPath: "/mnt/data"
      }
   }
```

In upon diagram, “/mnt/data” is the destination mount point of the docker container, per container “MountPath” is the source mount point of the docker container. And “MountPath” is where a volume is mounted to via CSI driver.

Note, depending on the capability of different storage systems, it may not allow a volume to be mounted onto multiple places, or it allows multiple mounts but doesn’t allow concurrency writes. This is defined as “access_mode” per CSI spec, YARN needs to do sanity checks to make sure the volume would not be abused. Moreover, the capability of the volume also determines what mount propagation flag can be used. User needs to ensure to use supported mount propagation flags for the volume mount, otherwise the container will fail to start due to mount failure.

When container is finished, the local mount on NM host needs to be disabled and removed. If container is crashed or killed, the cleanup also needs to be handled.

6. User Secrets

Credentials may be needed by CSI driver to interact with storage systems. This is defined as **Secrets** in the CSI spec. Secrets are key-value pairs that encoded sensitive information, e.g username=bob, password=abc123. The content might be encrypted depending on how CSI driver implements it. These info, if required, will also be specified in attributes:

```
yarn.io/s3-volume {
  size : 100,
  unit: Gi,
  Attributes : {
    driver: “s3-csi-driver”,
    mountPoint: “/mnt/amt”,
    accessMode: “readonly”,
    volumeId: “vol-01” // preprovisioned,
    Secret : “{
      username: “bob”,
      password: “abc123”
   }”
  }
}
```

According to the CSI spec, secrets is defined as a string string map, the keys can be vary from vendor to vendor, YARN is only responsible to pass them to CSI driver as long as they exist.