PURESTORAGE

Kubernetes, Persistent Volumes and the Pure Service Orchestrator (CSI Driver)

Simon Dodsley, Technical Director

Contents

Notices

Pure Storage, Inc.
650 Castro Street
Mountain View, CA 94041

Executive Summary

Over the last few years, there has been a dramatic change in the container ecosystem that is changing both the design and deployment methodologies for software. With the growth of microservices and the equivalent explosion in the number of containers running these, the management of clusters of containers has changed and now requires tools that have capabilities to deal with scheduling, load balancing, resource monitoring, scaling and job isolation of these complex environments. Not only do we need tools that can perform these roles, but must also be production-ready. One of the most complex production environments in the world has been run and managed by Google® over the last 15 years. For the last 10 years these environments have been largely run in Linux® containers, and to help them manage the complexities of this they developed their own container management system, called Borg.

In 2015 Google released to the world the Open Source project called Kubernetes®, frequently referred to as k8s™, which has its root firmly embedded in the Borg program, building on this and addressing some of the issues seen within Borg internally. Kubernetes is now one of the leading container orchestration tools and has even been embraced by many other Open Source software projects to either help manage and deploy themselves or to use these other projects to help deploy Kubernetes itself. Examples include OpenStack and OpenShift Origin. It should also be noted that many companies already heavily use Kubernetes and many cloud service providers have offerings that are based on this platform.

This white paper describes Kubernetes, its concepts, terminology and capabilities, and will then look at the role Pure Storage can play in Kubernetes deployments, specifically around persistent storage for stateful applications, specifically using the Pure Service Orchestrator.

Audience

This white paper is designed for developers, system architects and storage administrators who are evaluating or deploying Kubernetes and are interested in persistent storage within their deployment. Whilst this document will focus on Pure Storage products, the concepts discussed and implemented can be applied to other storage vendor products.

Whilst there is an assumption that the reader understands Kubernetes, this whitepaper will provide a high-level overview including concepts and terminology, before covering more detailed storage related concepts. Some knowledge of Pure Storage products would also be useful.

Kubernetes Terminology and Concepts

Kubernetes can be a complex application, especially for a beginner, so let's look at some of its basic terminology and concepts before we get deeper into the storage-centric content of this whitepaper.

Cluster

A cluster is a set of either physical or virtual machines on which you are going to run and manage your applications upon. In the world of Kubernetes, all machines are managed as either a cluster or set of clusters. These clusters consist of at least one master and multiple worker machines. All cluster members run the Kubernetes orchestration system.

Node

This is a machine unit, either physical or logical, which is part of a cluster and upon which you run your applications. Previously this has been referred to as a `minion` but is usually referred to as a worker node.

Pod

This is a group of one or more containers and their shared storage and networks. These resources are always co-located and co-scheduled and run in a shared context, in other words on a single node. Containers that run in the same pod share the same IP address and communicate with each other using `localhost` and other standard inter-process communication systems. Here multiple containers, in the same pod, are implicitly linked to the same application and are referred to as a master container and sidecar containers. If you deploy a single container per node then you can replace the word pod with container in your definitions.

Labels

Labels are key/value pairs that are given to objects, such as pods, and are used to specify attributes of the objects that are meaningful and relevant to users. These allow users to map their own organizational structures onto objects. Many objects can have the same or multiple labels. Example labels could be:

- `"release" : "stable","release" : "canary"`
- `"environment" : "dev","environment" : "qa","environment" : "production"`
- `"tier" : "frontend","tier" : "backend","tier" : "cache"`
- `"partition" : "customerA","partition" : "customerB"`
- `"track" : "daily","track" : "weekly"`

By using labels, a user can easily identify a set of objects using a label selector.

Selector

A selector is a core concept in Kubernetes and allows a user to identify a set of objects based on the labels that have been assigned to them. Selectors can be either equality-based or set-based and can be made of multiple comma-separated requirements which work using the logical AND operator. Using the examples above a user could employ an equality selector to find all objects that belong to a specific customer:

```
partition = customerA
```

or to locate all stable releases of objects:

```
release = stable
```

Replication Controller

A Replication Controller, sometimes abbreviated to RC, ensures that a certain number of pods are running at any one time. It is useful for enabling certain numbers of pods for scaling or to ensure there is always at least one pod. A Replication Controller can also terminate pods if too many are running. They are useful to ensure that pods are restarted in the case any fail, are restarted or are terminated. It is recommended to use Replication Controllers even if your application only uses a single pod.

Replica Set

A Replica Set defines how many replicas of a pods will be running and is the next generation Replication Controller. The only difference between the two is selector support. Replica Sets support both selector types, where RCs only supported equality-based selectors.

Service

A service is an abstraction which defines a logical set of pods and policies for accessing them, such as an IP address. As pods are mortal and have lifecycles, the service allows users to access their application across pod failures, node restarts, etc. that the RC may use to dynamically bring things back to a required state.

Names

Every object in Kubernetes is unambiguously identified by a name and a unique UID. Names are usually defined by clients and are unique, however, if an object is deleted a different object can then be given the same name.

Namespaces

Kubernetes can support multiple virtual clusters on the same physical cluster of nodes and these are referred to as namespaces. These can be used where there are numerous teams using the same physical infrastructure but need to keep their own resources separate. On install Kubernetes has three namespaces: `default`, `kube-system` and `kube-public`.

Images

Images are the key building blocks of any Kubernetes infrastructure. Currently, k8s only supports Docker images. These images are essentially the micro-services or applications that are run within a container and an image can consist of many files which can encapsulate an application. The Docker images are retrieved from a registry and then deployed and managed by Kubernetes.

Registry

A registry is a location where container images are stored for Kubernetes to download from. There are several locations that a registry can reside, either in the public domain, such as the Docker Public Registry, private cloud domains such as Google Private Registry or Amazon EC2 Container Registry or in the form of a local private registry located within an organization's own infrastructure.

Functionalities of Kubernetes

Whilst any container orchestration tool needs to be able to run and schedule jobs, there is now a growing requirement to effectively manage those containers in large, production-ready clusters, possibly across multiple datacenters. Kubernetes has this ability and here we will look at some of those requirements and some of the concepts used to perform these. Note that this is a quick summary and more detailed descriptions can be found in the Kubernetes documentation set.

Management of Secrets

A secret in the context of Kubernetes is an object that contains a small amount of sensitive data, such as a password, token, or key. This information could be retained in a container image or pod specification, but this could then make it vulnerable to security breaches. By encoding secrets in objects, Kubernetes allows them to be referred to without exposing the information to accidental exposure and to ensure that the secret is consistent for all resources that need to access it across the cluster.

Application Health

There are two types of health checks that can be applied to applications, these being the liveness and readiness probes. These can greater help with the stability of applications. The liveness probe checks to ensure that an application is running. This is usually and simply done by checking an HTTP endpoint. These probes can have parameters set to delay the initial check to cater for applications that take time to start, and how long to wait for a response in case the application is under heavy load. The readiness probe can be used to check an application is able to serve requests of it, such as an application requiring both a database to be running and memcached for example. If the readiness probe fails then the pod can be removed from the service and even terminated gracefully using hooks to ensure certain activities complete before the container is stopped.

Co-location of Related Processes

Where an application is made up of multiple components these should be tightly coupled to each other, allowing them to easier communicate and used shared storage. Kubernetes can be made to schedule these different components, each an image, to be co-located in the same pod.

Management and Scaling

One of the most important functions of any container management tool is its ability to replace a failing container with a new one, ensuring that the correct number of replicas are maintained in the environment. For example, Kubernetes uses Replication Controllers to maintain the correct number of replicas for a container, but it can also auto scale the number of replicas depending on different criteria, such as CPU utilization.

Load Balancing

Every application that has a scaling requirement also needs to be load balanced. Kubernetes achieves this by dynamically providing an ingress based load-balancing service. This service masks the underlying pods and exposes them as a single entity, usually as a single IP address or DNS name.

Application Upgrades

In this always-on world, users expect applications to be available always and therefore applications need to have new versions deployed seamlessly and potentially numerous times a day. Kubernetes allows this to happen using rolling updates. These allow deployments to take place with zero downtime by using load balancing to publicly expose pods that are available and not in the process of being upgraded.

Resource Monitoring

Traditional applications only required the monitoring of the application and the hosts they ran on. Now we additionally need to monitor the containers and the orchestration tools as well. These four layers of

monitoring are all carried out by different parts of Kubernetes and aggregated into Heapster, whose data can then be visualized by tools such as Grafana.

Log Management

Analyzing log data can be critical to understanding what is occurring in a Kubernetes cluster. There are several internal components to fetch and analyze data using log library and you can use the k8s CLI (`kubectl`) to get log data from containers. All this information can be sent to an ELK stack for further analysis and viewing.

Data and Storage

Historically applications in containers were ephemeral, short-lived application whose data was lost on container failure or restart. With the introduction of persistent storage for Kubernetes containers applications can now have access to data from other applications, or to stateful data from a container failure, container relocation, etc. A later section of this document will cover Kubernetes storage in greater detail.

Kubernetes Components

In this section, we will look at the main components of a Kubernetes cluster, some parts of which are optional, and some mandatory for the correct functioning of the environment.

Master Node

The master node is where all the administrative tasks of the cluster are made and is the command and control plane for Kubernetes. This node makes all the decisions about the cluster, such as scheduling, detecting and responding to cluster events, etc. The main components of the master node are:

API Server

The API server controls all the REST commands used to control the cluster by processing them, validating them and finally by executing them. The result of the REST calls is then held in the cluster data store.

Cluster Data Store

Kubernetes uses `etcd` for its cluster data store. This is a simple, strong, distributed, consistent key/value store and is used as a persistent store for all API objects within the cluster. It can be considered the "source of truth" for the cluster and holds such information as pod and service details and state, namespace and replica information, etc.

Scheduler

The deployment of pods and services onto nodes is handled by the scheduler component. This holds information regarding the resources available in cluster members, as well as resources required for a configured service which helps it decide where to deploy specific services.

Controller Manager

This is a daemon called `controller-manager`, also known as the "kube-controller manager". It used the API server to watch the shared state of the cluster and runs all controllers for routine tasks in the cluster. These include Node Controllers, Replication Controllers, Endpoint Controllers, and Service Account and Token Controllers.

Dashboard (optional)

There is an optional web UI that can be installed on the master node that allows Kubernetes users to easily interact with the API server.

Worker Nodes

Where the master node manages the whole cluster, a worker node, sometimes referred to as a minion, runs the actual containers and provide the Kubernetes runtime environment, including networking and storage. The main components of a worker node are:

Docker

Docker runs on all worker nodes and runs the configured pods. It performs all the tasks related to downloading images and starting containers.

Kubelet Service

`kubelet` is the primary agent process in a worker node. It watches the API server for pods that have been assigned to its node and is also responsible for reporting back to the master node the status of pods that are running on its node. It also communicates with `etcd` to get information about existing services and write details of new ones.

Proxy Service

This service is run by `kube-proxy` and acts as a network proxy and load balancer for a service on the worker node. It manages all the network routing for TCP and UDP packets to a service.

Command Line Utility

The command line tool that allows users to interact with the API server and send commands to the master node is `kubectl`.

Kubernetes Storage

As mentioned previously containers, historically, used only ephemeral data storage and therefore when a container stopped, failed, or restarted, the data associated with the application was lost.

With the development of applications and use cases that required data to be made available across container restarts, container relocations and across application micro-services, there developed a requirement for persistent storage.

Within the day to day usage of a Kubernetes cluster users, such as developers, may require the use of persistent storage for their pod and they would request this from Kubernetes using a declarative mechanism

such as JSON or YAML (we will cover what these look like later). The Kubernetes master will then check which worker node has the appropriate storage resource available and allocate it to the developer's pod using a process called dynamic provisioning.

Before dynamic provisioning became GA in the Kubernetes v1.6, an administrator of the cluster needed to pre-create underlying persistent storage for the cluster from storage providers. With dynamic provisioning, this is now fully automated.

Here we look at the main concepts of using persistent storage in Kubernetes and how they are implemented and accessed.

Storage Class

A `StorageClass` is a resource object within Kubernetes that is essentially a blueprint for storage that abstracts away the underlying storage provider. These are cluster-wide objects that need to be created by a cluster administrator. Different storage classes can link to different storage providers, thereby allowing storage with different capabilities to be made available.

The `StorageClass` resource must have a provisioner that determines what volume plugin is used for creating volumes when requested. Example internal provisioners are NFS, iSCSI, FC, and OpenStack Cinder. There are also external CSI drivers that can be installed to enable 3rd party storage backends to be used within a StorageClass. The following example shows the YAML definition to create a new StorageClass.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: pure-block
  labels:
    kubernetes.io/cluster-service: "true"
provisioner: pure-provisioner
parameters:
  backend: block
```

In this example, we are creating a resource based on the Pure Storage provisioner, specifically from a block-based Pure Storage backend. A file-based backend is also available from Pure Storage.

To check which StorageClasses are installed in a cluster using the '`kubectl get sc`' command.

Persistent Volumes

When a Storage Class has been defined a persistent volume (PV) can then be created using this class. This volume is available cluster-wide and has a lifecycle that is separate to any pod that may wish to use it.

For an administrator to create a PV the declarative YAML file contents would look like this:

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: pure-pv-volume
spec:
```

```
  storageClassName:  pure-block
  capacity:
    storage: 10Gi
  accessModes:
   - ReadWriteOnce
  hostPath:
    path: "/tmp/data"
```

From this file, you then create the PV using the command:

```
# kubectl create -f <YAML filename>.yaml
```

And to then view details about the PV:

```
# kubectl get pv pure-pv-volume
```

Persistent Volume Claims

The persistent volume claim (PVC) is the actual request by a user to access a PV and have it assigned to their pod. The PVC will define the size and access mode of the PV for the pod.

For a user to issue a PVC they need to create a declarative YAML definition as shown below:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pure-volume
spec:
  accessModes:
   - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName:  pure-block
```

From this file you then create the PV using the command:

```
# kubectl create -f <YAML filename>.yaml
```

And to then view details about the PVC:

```
# kubectl get pvc pure-volume
```

In this example, we are creating a 10GiB block storage volume using the Pure Storage provisioner provided by a FlashArray.

When using dynamic provisioning there is no requirement to create the PV as the act of creating the PVC will automatically create the required volume and when the claim is deleted the volume will also be deleted.

Introduction to Pure Service Orchestrator

Since 2017 Pure Storage has been building seamless integrations with container platforms and orchestration engines using the plugin model, allowing persistent storage to be leveraged by environments such as Kubernetes.

As adoption of container environments move forward the device plugin model is not sufficient to deliver the cloud experience developers are expecting. This is amplified by the fluid nature of modern containerized environments, where stateless containers are spun up and spun down within seconds and stateful containers have much longer lifespans. Where some applications require block storage, whilst others require file storage, and a container environment can rapidly scale to 1000s of containers. These requirements can easily push past the boundaries of any single storage system.

Pure Service Orchestrator was designed to provide a similar experience to your developers that they expect they can only get from the public cloud. Pure Service Orchestrator can provide a seamless container-as-a-service environment that is:

- **Simple, Automated and Integrated:** Provisions storage on demand automatically via policy, and integrates seamlessly enabling DevOps and Developer friendly ways to consume storage
- **Elastic:** Allows you to start small and scale your storage environment with ease and flexibility, mixing and matching varied configurations as your Swarm environment grows/
- **Multi-protocol:** Support for both file and block
- **Enterprise-grade:** Deliver the same Tier1 resilience, reliability and protection that your mission-critical applications depend upon, for stateful applications in your Kubernetes clusters
- **Shared:** Makes shared storage a viable and preferred architectural choice for the next generation, containerized datacenters by delivering a vastly superior experience relative to direct-attached storage alternatives.

Pure Service Orchestrator integrates seamlessly with your Kubernetes orchestration environment and functions as a control-plane virtualization layer that enables container-as-a-service rather than storage-as-a-service.

Using Pure Service Orchestrator for K8s Persistent Storage

As we have seen above we can use Pure Storage backends to provide PVs for PVCs issued by developers, although we only showed the block storage StorageClass, which uses a Pure Storage FlashArray as the backend storage provider. There is also a file-based StorageClass available, *pure-file*, that uses the Pure Storage FlashBlade™ as the backend storage provider to provide ReadWriteMany (RWX) persistent volumes.

To make these StorageClasses available to a Kubernetes cluster there is a requirement to install the Pure Storage Kubernetes CSI driver and dynamic provisioner. This is simply done using of the Pure Service Orchestrator which uses Helm to install and configure all the required Kubernetes settings, including StorageClasses.

Pure Service Orchestrator Installation

Installation and configuration of the Pure Service Orchestrator is simple and requires only a few steps, which are described in the Pure Storage Helm Chart GitHub repository.

There are a couple of actions that need to be performed on every k8s worker node in your cluster before performing the installation:

- ensure the latest multipath software package is installed and enabled
- ensure the `/etc/multipath.conf` file exists and contains the Pure Storage stanza as described in the Pure Storage Linux Best Practices.
- If using a FlashBlade as a storage appliance, ensure that the latest NFS tools appropriate to the operating system being used are installed.
- If using a FlashArray as a storage appliance, ensure that the latest iSCSI or fibre channel initiator software is installed.

PSO Installation (Using Helm)

The Pure Service Orchestrator manages the installation of all required files across your Kubernetes cluster by using a DaemonSet to perform this cross-node installation. The DaemonSet runs a pod on each node in the cluster, which copies the required files in the right path on the host for the kubelet to access. It will keep the config updated, and ensure that files are installed safely.

Installation of the Pure Service Orchestrator for Kubernetes requires that you have Helm installed on your Kubernetes cluster. After you have installed the Helm binaries you will need to run the `helm init` if you are using Helm 2 (not required for Helm 3) and then command perform the following steps:

- Add the `pure` repo to Helm

```
helm repo add pure http://purestorage.github.io/helm-charts
helm repo update
# Helm 2
helm search pure-csi
# Helm 3
helm search repo pure-csi
```

- Ensure you have a service account with cluster-admin role for Helm. If you issue the `helm list` command and receive the following error

```
Error: configmaps is forbidden: User "system:serviceaccount:kube-system:default" cannot list
configmaps in the namespace "kube-system"
```

then fix this with

```
# kubectl create clusterrolebinding add-on-cluster-admin --clusterrole=cluster-admin --
serviceaccount=kube-system:default
```

- Update configuration file

  To enable the Pure Service Orchestrator for Kubernetes to communicate with your Pure Storage backend array it is required to update the configuration file to reflect the access information for the backend storage providers. The file is called `values.yaml` and need to contain the management IP address of the backend devices, together with a valid, privileged, API token for each device, together with an NFS Data VIP address for each FlashBlade.

  Take a copy of the `values.yaml` provided by the Helm Chart and update the `arrays` parameters in the configuration file with your site specific information as shown in the following example:

```
arrays:
  FlashArrays:
    - MgmtEndPoint: "1.2.3.4"
      APIToken: "a526a4c6-18b0-a8c9-1afa-3499293574bb"
    - MgmtEndPoint: "1.2.3.5"
      APIToken: "b526a4c6-18b0-a8c9-1afa-3499293574bb"
  FlashBlades:
    - MgmtEndPoint: "1.2.3.6"
      APIToken: "T-c4925090-c9bf-4033-8537-d24ee5669135"
      NfsEndPoint: "1.2.3.7"
    - MgmtEndPoint: "1.2.3.8"
      APIToken: "T-d4925090-c9bf-4033-8537-d24ee5669135"
      NfsEndPoint: "1.2.3.9"
```

  Ensure that the values you enter are correct for your own Pure Storage devices.

  There are multiple configuration options available for PSO through settings in the configuration file. More details can be found here: https://github.com/purestorage/helm-charts/blob/master/pure-csi/README.md

- Install the plugin

  It is advisable to perform a 'dry run' installation to ensure that your `values.yaml` file is correctly formatted:

```
# Helm 2
helm install --name pure-storage-driver pure/pure-csi -f <your_own_dir>/<your_own_values>.yaml --
dry-run --debug

# Helm 3
helm install pure-storage-driver pure/pure-csi -f <your_own_dir>/<your_own_values>.yaml --dry-run
--debug
```

  Run the actual install.

```
# Helm 2
helm install --name pure-storage-driver pure/pure-csi -f <your_own_dir>/<your_own_values>.yaml

# Helm 3
helm install pure-storage-driver pure/pure-csi -f <your_own_dir>/<your_own_values>.yaml
```

  The values set in your own YAML will overwrite the ones in the default, provided, file, but the `--set` option can also take precedence over any value in either YAML, for example:

```
# Helm 2
helm install --name pure-storage-driver pure/pure-csi -f <your_own_dir>/<your own values>.yaml --
set flasharray.sanType=FC,namespace.pure=k8s_xxx

# Helm 3
helm install pure-storage-driver pure/pure-csi -f <your_own_dir>/<your own values>.yaml --set
flasharray.sanType=FC,namespace.pure=k8s_xxx
```

However, it is recommended to use the `values.yaml` file rather than the `--set` option for ease of use, especially should modifications are required to your configuration in the future.

PSO Installation (Using Operators)

With the advent of Kubernetes 1.13, there is now a second installation process that can be used which negates the need to install and use Helm (and Tiller). This is using the Operator framework.

The installation of the Pure Service Orchestrator for Kubernetes using the Operator framework is performed as follows:

- Obtain the Operator installation files

  Pull the latest Operator files from the Pure Storage GitHub repository by creating a clone:

```
# git clone --branch 2.5.5 https://github.com/purestorage/helm-charts.git
# cd helm-charts/operator-csi-plugin
```

  This example is getting the 2.5.5 version of the code, but make sure you always get the latest released version of the driver. Ensure you only use the code in the directory mentioned above.

  Get a copy of the matching `values.yaml` using the following command:

```
# wget https://raw.githubusercontent.com/purestorage/helm-charts/2.5.5/pure-csi-plugin/values.yaml
```

  Update this file to ensure the configuration matches your specific environment. Ensure that the `arrays` parameters in the configuration file has your array information populated correctly. For example:

```
arrays:
  FlashArrays:
    - MgmtEndPoint: "1.2.3.4"
      APIToken: "a526a4c6-18b0-a8c9-1afa-3499293574bb"
    - MgmtEndPoint: "1.2.3.5"
      APIToken: "b526a4c6-18b0-a8c9-1afa-3499293574bb"
  FlashBlades:
    - MgmtEndPoint: "1.2.3.6"
      APIToken: "T-c4925090-c9bf-4033-8537-d24ee5669135"
      NfsEndPoint: "1.2.3.7"
    - MgmtEndPoint: "1.2.3.8"
      APIToken: "T-d4925090-c9bf-4033-8537-d24ee5669135"
      NfsEndPoint: "1.2.3.9"
```

  Ensure that the values you enter are correct for your own Pure Storage devices.

There are multiple configuration options available for PSO through settings in the configuration file. More details can be found here: https://github.com/purestorage/helm-charts/blob/master/pure-csi/README.md

- Install the plugin

```
# ./install.sh -namespace=pso-operator -orchestrator=k8s -f values.yaml
```

Note that `orchestrator` parameter can be set to `openshift`, should you want to install in a Red Hat OpenShift cluster. The `namespace` parameter is the name of the namespace that PSO will be installed in.

Checking Correct Installation

After running the installer there are a few things we can check to ensure that the plugin has installed correctly.

- Check StorageClasses

```
# kubectl get sc
NAME             TYPE
pure             pure-provisioner
pure-block       pure-provisioner
pure-file        pure-provisioner
```

- Check CSI driver running on each worker node

```
# kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
pure-csi-4mkfw                      1/1       Running   0          2h
pure-csi-527zn                      1/1       Running   0          2h
pure-csi-7n9cf                      1/1       Running   0          2h
pure-csi-8gfbz                      1/1       Running   0          2h
pure-csi-9wh9d                      1/1       Running   0          2h
pure-csi-g7xvh                      1/1       Running   0          2h
pure-csi-jxxbn                      1/1       Running   0          2h
pure-csi-w9x74                      1/1       Running   0          2h
pure-provisioner-2785090122-c69xx   1/1       Running   0          2h
```

If you have installed using the Operator process you will see an additional pod:

```
pso-operator   pso-operator-6876f95888-dkkwx    1/1     Running   2          2h
```

- Check Deployments

```
# kubectl get deployments
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
pure-provisioner   1         1         1            1           2h
```

If you have installed using the Operator process you will see an additional deployment:

```
pso-operator   pso-operator              1/1     1            1           2h
```

- Check Custom Resource Definition (Operator install process only)

```
# kubectl get crd
NAME                          CREATED AT
psoplugins.purestorage.com    2019-05-31T12:38:43Z
```

Simple Proof Point of PVs from Pure Storage

To show that data on a PV is retained and can be made available to the same application on different nodes after a pod restart, we can do a simple test using an Nginx application pod.

First, we create our PVC, which the dynamic provisioner will use to create a PV (the name of which will match the volume name on the Pure Storage FlashArray, or share name on a FlashBlade, with a `k8-` prefix):

```
# cat <<EOF >> nginx-pvc.yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  # Referenced in nginx-pod.yaml for the volume spec
  name: pure-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: pure
EOF
# kubectl create -f nginx-pvc.yaml
# kubectl get pvc
NAME           STATUS    VOLUME                                     CAPACITY   ACCESSMODES   STORAGECLASS   AGE
pure-claim     Bound     pvc-bf1c9108-e4cd-11e7-ad7c-ecf4bbe57354   10Gi       RWO           pure           40m
```

Secondly, we will create a simple Nginx pod using the PVC as a mount point within the pod:

```
# cat <<EOF >> nginx-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: default
spec:
  # Specify a volume that uses the claim defined in nginx-pvc.yaml
  volumes:
  - name: pure-vol
    persistentVolumeClaim:
        claimName: pure-claim
  containers:
  - name: nginx
    image: nginx
    # Configure a mountpoint for the volume we defined above
    volumeMounts:
    - name: pure-vol
      mountPath: /data
     ports:
    - containerPort: 80
    securityContext:
      privileged: true
EOF
# kubectl create -f nginx-pod.yaml
```

Looking at the pod description we can see the host the pod is running on and that the PVC is linked to a mount point:

```
# kubectl describe pod nginx
Name:           nginx
Namespace:      default
Node:           sn1-pool-c07-08.puretec.purestorage.com/10.21.200.118
Start Time:     Mon, 20 Jan 2020 07:29:09 -0800
Labels:         <none>
Annotations:    openshift.io/scc=privileged
Status:         Running
IP:             10.129.2.15
Containers:
  nginx:
    Container ID:       docker://9a82f70414526b20e0bcf3b9bcd36359d711d58a44f7484cf942601ad3197135
    Image:              nginx
    Image ID:           docker-pullable://docker.io/nginx@sha256:2ffc60a51c9d658594b63ef5acfac9d92f4e1550f633a…
    Port:               80/TCP
    State:              Running
      Started:          Mon, 20 Jan 2020 07:29:13 -0800
    Ready:              True
    Restart Count:      0
    Environment:        <none>
    Mounts:
      /data from pure-vol (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-7t562 (ro)
Conditions:
  Type          Status
  Initialized   True
  Ready         True
  PodScheduled  True
Volumes:
  pure-vol:
    Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace)
    ClaimName:  pure-claim
    ReadOnly:   false
```

We can now open a bash terminal to the pod and interrogate the environment and create a file in the mount point of the PV:

```
# kubectl exec -ti nginx – bash
root@nginx:/# df
Filesystem                                                     1K-blocks     Used Available Use% Mounted on
/dev/mapper/docker-253:0-101137330-5f10f40cc5f82bb6b32cccb 10475520   149044  10326476   2% /
tmpfs                                                          65993132        0  65993132   0% /dev
tmpfs                                                          65993132        0  65993132   0% /sys/fs/cgroup
/dev/mapper/3624a9370bde00b13de084ce300012f55                  10475520    32944  10442576   1% /data
/dev/mapper/rhel_sn1--r620--c07--08-root                       52403200 10423784  41979416  20% /etc/hosts
shm                                                               65536        0     65536   0% /dev/shm
tmpfs
root@nginx:/# cd /data
root@nginx:/data# ls
root@nginx:/data# date > temp-data
root@nginx:/data# cat temp-data
Mon Jan 20 15:34:49 UTC 2020
root@nginx:/data# exit
exit
#
```

We can now delete the pod and recreate it. Checking that the pod has come up on a different node, we can now open a new bash terminal and check the contents of the previously created file:

```
# kubectl delete pod nginx
pod "nginx" deleted
# kubectl create -f nginx-pod.yaml
pod "nginx" created
# kubectl describe pod nginx
Name:           nginx
Namespace:      default
Node:           sn1-pool-c07-07.puretec.purestorage.com/10.21.200.117
Start Time:     Mon, 20 Jan 2020 07:57:53 -0800
```

```
Labels:         <none>
Annotations:    openshift.io/scc=privileged
Status:         Running
IP:             10.128.2.65
Containers:
  nginx:
    Container ID:       docker://1950f4c09ac788640a4bd281547f366e891a015c59eb1d2435eeefcfcd179463
    Image:              nginx
    Image ID:           docker-pullable://docker.io/nginx@sha256:2ffc60a51c9d658594b63ef5acfac9d92f4e1550f633…
    Port:               80/TCP
    State:              Running
      Started:          Mon, 20 Jan 2020 07:57:58 -0800
    Ready:              True
    Restart Count:      0
    Environment:        <none>
    Mounts:
      /data from pure-vol (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-7t562 (ro)
Conditions:
  Type          Status
  Initialized   True
  Ready         True
  PodScheduled  True
Volumes:
  pure-vol:
    Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace)
    ClaimName:  pure-claim
    ReadOnly:   false
<…>
<…>
<…>
# kubectl exec -ti nginx – bash
root@nginx:/# cat /data/temp-data
Mon Jan 20 15:34:49 UTC 2020
root@nginx:/# exit
exit
#
```

Updating your Storage Configuration

No environment is static and use of a Kubernetes cluster will inevitably grow as more applications and users use the resource. As part of this growth there could be a demand for additional backend storage, be that adding more FlashArrays for greater capacity, or adding FlashBlades into a block only environment to facilitate shared volumes across containers and applications, or just adding new labels to existing storage to allow for more granular control of storage placement.

Changing or adding Pure Storage backend devices in an existing deployment of the Pure Service Orchestrator is seamless and simple.

Update your values YAML file with your new FlashArrays or FlashBlades, or add new labels to existing devices, then update the Kubernetes configuration maps using the following command when using Helm:

```
# helm upgrade pure-storage-driver pure/pure-csi -f <your_own_dir>/<your_own_values>.yaml
```

If you used the `--set` option when initially deploying the plugin you must use the same option again, unless these have been incorporated into your latest YAML file.

If you installed using the Operator process, the upgrade process is performed using the following command:

```
# ./update.sh -f <your_own_dir>/<your_own_values>.yaml
```

Conclusion

As applications evolve from monolithic architectures to those based on more agile microservices, operations teams need to meet the needs of these developers on both a server and storage based front.

With the evolution of containers these, initially, seemed to solve the infrastructure solution for developers, but as applications that require a more persistent nature are being developed and ported to containers, the ephemeral and stateless nature of containers has caused issues.

By providing the ability to provide external persistent volumes for Kubernetes managed containers, the Pure Service Orchestrator plugin now allows the DevOps community to fully utilize the Tier 1 capabilities of the FlashBlade and FlashArray, including their state of the art data reduction capabilities, solid-state speed and seamless expansion and upgrade functionality. Together with the simplicity inherent in Pure Storage products and the simple integration with Kubernetes environments, this plugin has expanded the capabilities of the DevOps communities deployment and development functions to fully maximize their infrastructure solutions.

About the Author

As Technical Director of New Stack, Simon is responsible for managing the direction of Pure Storage pertaining to Open Source technologies including OpenStack, Containers, and associated orchestration and automation toolsets. His responsibilities also include developing best practices, reference architectures and configuration guides.

With over 30 years of storage experience across all aspects of the discipline, from administration to architectural design, Simon has worked with all major storage vendors' technologies and organisations, large and small, across Europe and the USA as both customer and service provider.

Blog: http://blog.purestorage.com/author/simon

PURESTORAGE

Pure Storage, Inc.
Twitter: @purestorage
www.purestorage.com

650 Castro Street, Suite #260
Mountain View, CA 94041

T: 650-290-6088
F: 650-625-9667

Sales: sales@purestorage.com
Support: support@purestorage.com
Media: pr@purestorage.com
General: info@purestorage.com