

TECHNICAL WHITE PAPER

Containers as a Service

Integrate Kubernetes and Pure Storage Solutions.

Contents

Introduction	4
Containers as a Service (CaaS).....	4
CaaS Defined.....	4
Components, Prerequisites, and Configuration.....	4
Pure Storage FlashArray™//X	4
FlashArray//X Technical Specifications.....	5
Pure Storage FlashArray//C.....	6
FlashArray//C Technical Specifications.....	6
Purity for FlashArray (Purity//FA 6).....	6
Pure Storage FlashBlade®	7
Meeting the Needs of Unified Fast File and Object for Modern Applications and Modern Data	7
Purity for FlashBlade (Purity//FB)	8
Pure1®	9
Pure1 Manage	9
Pure1 Analyze	9
Pure1 Support	9
Pure1 Meta	9
Evergreen™ Storage	9
Pure Service Orchestrator™	10
Kubernetes	11
High-level Design	11
Software Version Details	12
Compute	12
Networking	12
Deployment	13
Persistent Storage	13
Pure Service Orchestrator Installation	14

PSO Installation (using Helm3).....	14
Validate Correct Plugin Installation	16
Scaling Backend Storage	17
Simple Pod Deployment with a Pure Storage FlashArray Persistent Volume	17
Persistent Volume Claim.....	17
Application Deployment	19
Simple Multiple Pod Deployment with a Pure Storage FlashBlade Persistent Volume	22
Persistent Volume Claim.....	22
Nginx Application Pod	24
Additional Application Pod.....	26
Adding Nodes to a Kubernetes Cluster	28
Conclusion.....	28
Appendix: Application Examples	28
MongoDB.....	28
WordPress	31
About the Author	33



Introduction

This document provides a practical reference implementation to help integrate Pure Storage® products into the deployment of a bare-metal Kubernetes infrastructure. You can easily scale this underlying infrastructure to whatever size is required. This document assumes that you understand how to deploy a bare-metal Kubernetes solution and provides details only for Pure Storage integration pieces. Find links to details on Kubernetes deployments in the Appendix.

Containers as a Service (CaaS)

CaaS Defined

Containers as a Service is an implementation of container-based virtualization, where container engines, underlying compute servers, and orchestration toolsets are made available to users from a provider. The providers range from the big three cloud providers, down to private, on-premises, company-owned solutions.

CaaS can provide users with an architecture to enable DevOps teams the agility to automate ‘code check-in and go-live’ process for containerized solutions, which can significantly reduce the time to deploy and time to go-live into production for these applications. CaaS is for deploying applications where there is a requirement for more control over the components of applications and a requirement for developers to have a greater understanding of the build and run processes required by the application. For example, in a CaaS environment a developer who has written an application in, say, Python, needs to understand how to create an empty container image with a base filesystem and then move the code into the container locally. You might then have a requirement to compile the code, download dependencies, and finally create a Docker image. Only when the image has been created can it be used in the CaaS platform.

Components, Prerequisites, and Configuration

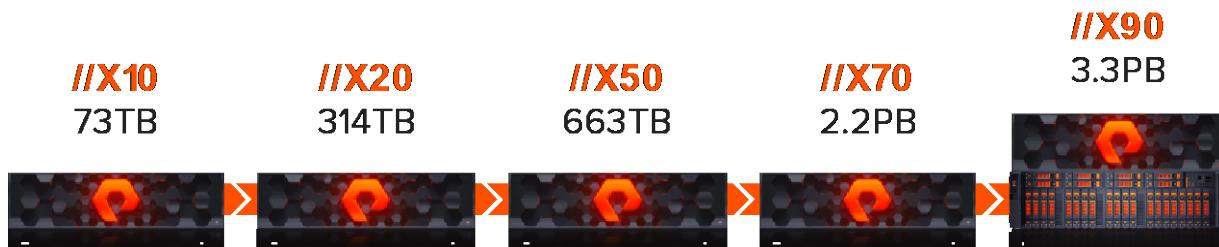
Pure Storage FlashArray™//X

FlashArray is the world’s first 100% all-flash end-to-end NVMe and NVMe-oF array, ideal for the most demanding enterprise performance requirements. FlashArray provides customers with a modern data experience, delivering breakthroughs in speed, simplicity, flexibility, and consolidation. It’s ideal for departmental to large-scale enterprise shared-storage deployments, high performance, and mission-critical applications. In a world of fast, pervasive networking, ubiquitous flash memory, and an evolving scale-out application architecture, Pure Storage’s FlashArray provides customers with both networked and direct-attached storage in a single, shared architecture. With latency as low as 150 µs, FlashArray brings new levels of performance to mission-critical business applications and databases.

FlashArray //X



From entry level to enterprise workloads, FlashArray//X lets your organization accelerate your most critical applications. FlashArray//X delivers major breakthroughs in performance, simplicity, and consolidation. It's ideal both for enterprise applications such as Oracle, SQL Server, and SAP, as well as cloud-native, web-scale applications such as MongoDB, Cassandra, Hadoop, and MariaDB. The FlashArray//X70 and //X90 support optional DirectMemory Cache, which uses Intel Optane storage class memory (SCM) to run database workloads at near-DRAM speeds. If extreme performance is a top priority, your organization can rely on FlashArray//X to deliver the low latency and high throughput end users demand.



FlashArray//X Technical Specifications

	Capacity*	Physical
//X10	Up to 73TB/66.2TiB effective capacity Up to 22TB/19.2TiB raw capacity	3U, 640-845 Watts (nominal–peak) 95 lbs (43.1kg) fully loaded, 5.12"x18.94"x29.72" chassis
//X20	Up to 314TB/285.4TiB effective capacity Up to 94TB/88TiB raw capacity	3U, 741-973 Watts (nominal–peak) 95 lbs (43.1kg) fully loaded, 5.12"x18.94"x29.72" chassis
//X50	Up to 663TB/602.9TiB effective capacity Up to 20TB/18.6TiB raw capacity	3U, 868-1114 Watts (nominal–peak) 95 lbs (43.1kg) fully loaded, 5.12"x18.94"x29.72" chassis
//X70	Up to 2286TB/2078.9TiB effective capacity Up to 622TB/544.2TiB raw capacity	3U, 1084-1344 Watts (nominal–peak) 97 lbs (44kg) fully loaded, 5.12"x18.94"x29.72" chassis
//X90	Up to 3.3PB/3003.1TiB effective capacity Up to 878TB/768.3TiB raw capacity	3U-6U, 1160-1446 Watts (nominal–peak) 97 lbs (44kg) fully loaded, 5.12"x18.94"x29.72" chassis
Direct Flash Shelf	Up to 1.9PB effective capacity Up to 512TB/448.2TiB raw capacity	3U, 460–500 Watts (nominal–peak) 87.7 lbs (39.8 kg) fully loaded, 5.12"x18.94"x29.72" chassis

* Effective capacity assumes HA, RAID, and metadata overhead, GB-to-GiB conversion, and includes the benefit of data reduction with always-on inline deduplication, compression, and pattern removal. Average data reduction is calculated at 5-to-1 and does not include thin provisioning.



Pure Storage FlashArray//C

Pure Storage FlashArray//C lets you consolidate workloads and simplify storage with consistent all-flash performance at a lower TCO than hybrid storage. FlashArray//C provides a 100% NVMe all-flash foundation for capacity-oriented applications, test and development workloads, multi-site disaster recovery, and data protection at hybrid storage economics. Scale up to 5.2PB effective storage in just three- to nine-rack units. Maximize results and flexibility for high-capacity applications on-premises and easily connect to the cloud. With Pure Evergreen™, you can upgrade performance, capacity, and features over time without disruption.

FlashArray//C Technical Specifications

	Capacity	Physical
//C60-366	Up to 1.3PB effective capacity 366TB raw capacity	3U, 1000-1240 Watts (nominal–peak) 97.7 lbs (44.3kg) fully loaded, 5.12"x18.94"x29.72" chassis
//C60-494	Up to 1.9PB effective capacity 494TB raw capacity	3U, 1000-1240 Watts (nominal–peak) 97.7 lbs (44.3kg) fully loaded, 5.12"x18.94"x29.72" chassis
//C60-840	Up to 3.2PB effective capacity 840TB raw capacity	6U, 1480-1760 Watts (nominal–peak) 177 lbs (80.3kg) fully loaded, 10.2"x18.94"x29.72" chassis
//C60-1186	Up to 4.6PB effective capacity 1.2PB raw capacity	6U, 1480-1760 Watts (nominal–peak) 185.4 lbs (84.1kg) fully loaded, 10.2"x18.94"x29.72" chassis
//C60-1390	Up to 5.2PB effective capacity 1.4PB raw capacity	9U, 1960-2280 Watts (nominal–peak) 273.2 lbs (123.9kg) fully loaded, 15.35"x18.94"x29.72" chassis

Purity for FlashArray (Purity//FA 6)

The Pure Storage® Purity operating environment is the software-defined engine of Pure Storage FlashArray. Purity is the driver that enables Pure FlashArray products, powering FlashArray//X to deliver comprehensive data services for your performance-sensitive data-center applications, and FlashArray//C for your capacity-oriented applications. Purity's core technologies provide the speed, agility, and intelligence needed to simplify everything in your production environment. Its features set the pace for next-generation shared accelerated storage, from enterprise data services for all workloads to proven FlashArray 99.9999% availability and on average 10:1 total efficiency. And with the Pure Evergreen™ ownership model, your Pure as-a-Service includes new array features and improvements to Purity via non-disruptive upgrades. Purity implements communication protocols and delivers rich data services across all Pure FlashArray systems. Features including ActiveCluster™ for business continuity and ActiveDR for disaster recovery, QoS, vVols, NVMe-oF, Snap to NFS, Purity CloudSnap™, DirectMemory™ Cache, and EncryptReduce are all examples of valuable new features provided with non-disruptive Purity upgrades. All Purity storage services, APIs, and advanced data services are built-in and included with every array. These technologies are driving the next-generation performance and industry-leading resiliency of Pure solutions.



Pure Storage FlashBlade®

Pure Storage FlashBlade is a Unified Fast File and Object (UFFO) storage platform that enables organizations to consolidate modern data intensive applications and cloud native operations onto a single scalable storage platform. FlashBlade allows organizations of all sizes to eliminate complex and inefficient infrastructure silos while delivering new levels of investment protection that support both private and hybrid cloud use cases. FlashBlade's unparalleled simplicity and multidimensional all-flash performance empower modern data and modern applications needs. Customers today are leveraging FlashBlade for real-time analytics, rapid recovery of backup data along with ransomware mitigation, and accelerated DevOps pipelines that are supporting the digital transformation priorities of business ranging from small municipalities to Fortune 100 enterprises across a diverse set of industries around the globe.

Meeting the Needs of Unified Fast File and Object for Modern Applications and Modern Data

FlashBlade delivers unprecedented performance, simplicity and consolidation with its massively distributed architecture that enables consistent performance for modern applications using NFS, S3/Object, SMB, and HTTP protocols. With FlashBlade UFFO, customers can scale out performance and capacity without scaling up complexity.

Simplicity	Performance	Consolidation
<ul style="list-style-type: none"> Evergreen architecture and non-disruptive upgrades Requires one-tenth the set-up time and effort of competitors' environments Low complexity and easy to manage Simplify lifecycle and data services management Reduces networking complexity and costly network switches/port 	<ul style="list-style-type: none"> Deliver linear, predictable storage performance while scaling up to tens of billions of files and objects Ability to run Fast File and Fast Object on-premises or in a hybrid or multi-cloud architecture Investment protection to grow and scale-out as necessary without stranding capacity or performance Provide multi-dimensional performance that spans across data sets and sizes 	<ul style="list-style-type: none"> Enables multiple applications to leverage the same platform for data needs instead of duplicating across silos or using point solutions Eliminates stranded storage capacity and performance. Disaggregates compute and storage to remove the complexity associated with DAS (Direct Attached Storage) Provide cloud-optimized efficiency for data and data services





Blade	Purity//FB	Fabric
Scale-Out DirectFlash + Compute Ultra-low latency, 8, 17, and 52TB capacity options that can be hot-plugged into the system for expansion and performance with the capability to scale from 7 to 150 blades non-disruptively.	Scale-Out Storage Software The heart of FlashBlade, implementing its scale-out storage capabilities, services, and management.	Software-Defined Networking Includes a built in 40Gb Ethernet fabric, providing a total network bandwidth of 320Gb/s for the chassis.
Power, Density, Efficiency		
FlashBlade delivers industry-leading throughput, IOPS, latency, and capacity – with up to 20x less space and 10x less power and cooling.		



Purity for FlashBlade (Purity//FB)

FlashBlade is built on the scale-out metadata architecture of Purity for FlashBlade, capable of handling 10s of billions of files and objects while delivering maximum performance, effortless scale, and global flash management. The distributed transaction database built into the core of Purity means storage services at every layer are elastic: simply adding blades grows system capacity and performance, linearly and instantly. Purity//FB supports S3-compliant object store, offering ultrafast performance at scale. In addition, FlashBlade offers File and Object replication services for site to site, as well as on-premises to public cloud via Object replication in AWS S3 in its native format. It also supports File protocols including NFSv3 and SMB, and offers a wave of new enterprise features, like snapshots, LDAP, network lock management (NLM), and IPv6, to extend FlashBlade into new use cases.



Pure1®

Pure1, our cloud-based management, analytics, and support platform, expands the self-managing, plug-n-play design of Pure all-flash arrays with the machine learning predictive analytics and continuous scanning of Pure1 Meta™ to enable an effortless, worry-free data platform.

Pure1 Manage

In the Cloud IT operating model, installing and deploying management software is an oxymoron: you simply log in. Pure1 Manage is SaaS-based, allowing you to manage your array from any browser or the Pure1 Mobile App – with nothing extra to purchase, deploy, or maintain. From a single dashboard, you can manage all your arrays, with full visibility on the health and performance of your storage.

Pure1 Analyze

Pure1 Analyze delivers accurate performance forecasting – giving you complete visibility into the performance and capacity needs of your arrays – now and in the future. Performance forecasting enables intelligent consolidation and unprecedented workload optimization.

Pure1 Support

Pure combines an ultra-proactive support team with the predictive intelligence of Pure1 Meta to deliver unrivalled support that's a key component in our proven FlashArray 99.9999% availability. Customers are often surprised and delighted when we fix issues they did not even know existed.

Pure1 Meta

The foundation of Pure1 services, Pure1 Meta, is global intelligence built from a massive collection of storage array health and performance data. By continuously scanning call-home telemetry from Pure's installed base, Pure1 Meta uses machine learning predictive analytics to help resolve potential issues and optimize workloads. The result is both a white glove customer support experience and breakthrough capabilities like accurate performance forecasting. Meta is always expanding and refining what it knows about array performance and health, moving the Data Platform toward a future of self-driving storage.

Evergreen™ Storage

Customers can deploy storage once and enjoy a subscription to continuous innovation via Pure's Evergreen Storage ownership model: expand and improve performance, capacity, density, and/or features for 10 years or more – all without downtime, performance impact, or data migrations. Pure has disrupted the industry's 3-5-year rip-and-replace cycle by engineering compatibility for future technologies right into its products.



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 no longer 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. Some applications in these environments require block storage, while others require file storage, and a container environment can rapidly scale to 1000s of containers. These requirements can quickly push past the boundaries of any single storage system. We designed Pure Service Orchestrator™ to provide your developers with a similar experience to what they expect from the public cloud. Pure Service Orchestrator can offer 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 Kubernetes 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 data centers by delivering a vastly superior experience relative to direct-attached storage alternatives.

Stateful: Complete with a fully managed cloud-native database to enable enhanced feature support and disaster recovery protection.

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



Kubernetes

Kubernetes is an Open Source system for managing containerized applications across multiple hosts, providing basic mechanisms for deployment, maintenance, and scaling of applications. The Open Source project is hosted by the Cloud Native Computing Foundation.

Kubernetes coordinates a highly available cluster of computers that are connected to work as a single unit. The abstractions in Kubernetes allow you to deploy containerized applications to a cluster without tying them explicitly to individual machines. To make use of this new model of deployment, applications need to be packaged in a way that decouples them from individual hosts: they need to be containerized. Containerized applications are more flexible and available than in past deployment models, where applications were installed directly onto specific machines as packages deeply integrated into the host. Kubernetes automates the distribution and scheduling of application containers across a cluster in a more efficient way. Kubernetes is an [open-source](#) platform and is production-ready.

High-level Design

The reference implementation used and described in this document consists of a six-node cluster, consisting of two kube-master hosts, with all six nodes being considered node hosts. The clustered etcd key-value store is run over three nodes. It is responsible for managing the entire cluster, where the node hosts run the applications within pods and communications between the kube-master nodes and the etcd system using APIs. This is shown in the figure below:

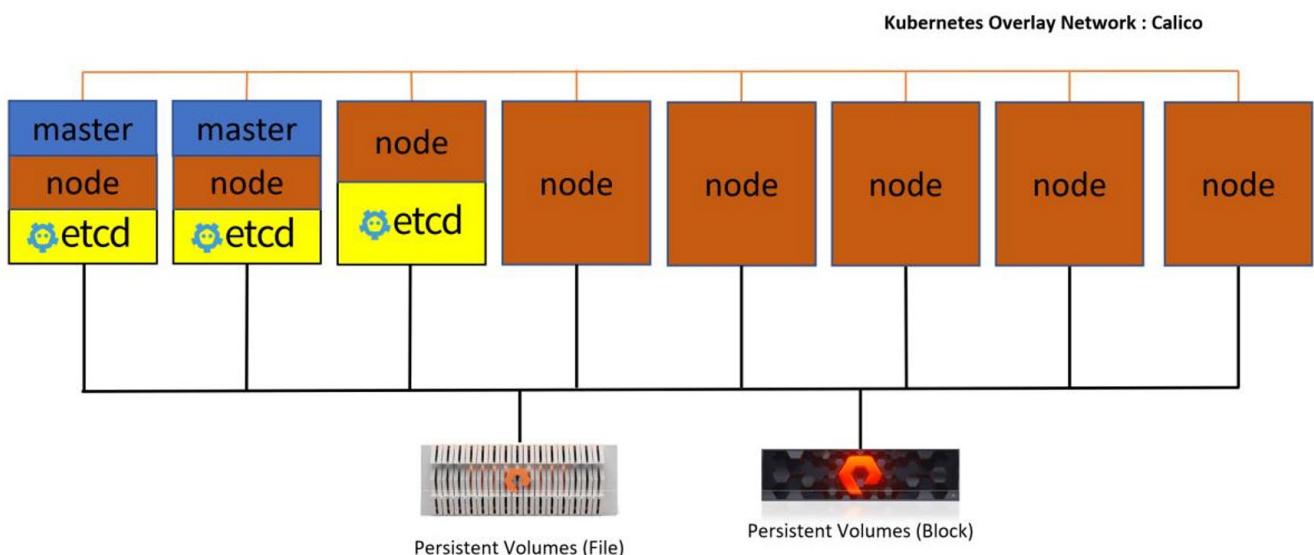


Figure 1. High-Level Architectural Design



Software Version Details

This table provides the installed software versions for the different components used in building this Reference Implementation.

Software	Version
Ubuntu 20.04	kernel 5.4.0-42-generic
Kubernetes	1.18.6
Docker	1.19.12
Helm	3.2.3
Pure Service Orchestrator	6.0.1

Compute

The role of this Reference Implementation is not to prescribe specific compute platforms for a Kubernetes cluster. Therefore, we refer to the servers being used in white-box terms. The servers used here have the following specifications:

- Intel® Xeon® E5-2630 v3 @ 2.40GHz
- 32 vCPU
- 128 GiB memory

Networking

From a networking perspective, the servers in use have the following connected network interfaces:

- 1 x 10GbE (management)
- 1 x 10GbE (iSCSI data plane)

There is no specific network hardware defined within this document, as this decision is dependent on the actual implementation performed by the reader. Within the Kubernetes networking layer, this implementation uses the Calico network plugin, the default provided by kubespray, although there are other network plugins available.

The networking communication between the Pure Service Orchestrator and backing storage devices requires that all cluster nodes have management plane access to all FlashArray and FlashBlade devices. FlashBlade data plane communication is performed using the NFS protocol whereas data plane communication between cluster nodes and FlashArrays is performed using an iSCSI network that can be either layer 2 or layer 3 depending on your network architecture. In this implementation, the iSCSI data plane is isolated from the management plane network, and jumbo frames are used end-to-end for the data plane. Fibre Channel is also a supported data plane protocol for FlashArrays, but this would require HBA cards to be installed in all cluster nodes and for zones to have been created between FlashArray FC ports and all cluster nodes prior to installing the Pure Service Orchestrator.



Deployment

While it is not in the scope of this document to go into detail on how to build a Kubernetes cluster, this deployment was implemented using the Kubernetes Incubator project, kubespray. If you decide to use kubespray as your deployment toolset, you are then recommended to perform the following tasks to ensure a smooth deployment on all cluster nodes:

- Ensure swap is disabled on all cluster nodes and the swap entry is removed from /etc/fstab
- Disable the firewalld software as this will interrupt the Kubernetes API communications within the cluster

To ensure that all FlashArray connections are optimal, it is necessary to install the latest **multipath-tools**, **open-iscsi** and **nfs-common** package, and then enable both the **multipathd** and **iscsid** daemons, to ensure they persist after any reboots. More details can be found in the [Pure Knowledge Base article on Linux Recommendations](#).

Note: *It is also advisable to implement the udev rules defined in the Knowledge Base article mentioned above to ensure optimal performance of your connected Pure Storage volumes.*

At this point, the deployment of the Kubernetes cluster can proceed using kubespray¹. By default, kubespray installs the Kubernetes Dashboard so you will want to grant the Dashboard Service Account Admin privileges² or create a user to access the dashboard³.

After completing the deployment of your cluster, it is necessary to install Helm⁴ as the Pure Storage plugin detailed below uses Helm Charts for deployment.

Persistent Storage

Within Kubernetes, we can use multiple Pure Storage backends to provide persistent storage in the form of Persistent Volumes for Persistent Volume Claims issued by developers.

The Pure Storage Kubernetes plugin provides both file- and block-based Storage Classes, provisioned from either FlashArray or FlashBlade storage devices. To make these Storage Classes available to your Kubernetes cluster, you must install the Pure Service Orchestrator in the form of the Pure Storage Kubernetes plugin.

¹ <https://github.com/kubernetes-sigs/kubespray>

² <https://github.com/kubernetes/dashboard/blob/master/docs/user/access-control/README.md#admin-privilages>

³ <https://github.com/kubernetes/dashboard/blob/master/docs/user/access-control/creating-sample-user.md>

⁴ <https://helm.sh/docs/intro/install>



Pure Service Orchestrator Installation

Installation and configuration of the Pure Service Orchestrator is simple and requires only a few steps, which are described in our GitHub repository for the Pure Storage Kubernetes plugin. However, 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 Linux Best Practices referenced above.

PSO Installation (using Helm3)

The Pure Service Orchestrator manages the installation of all required software 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 ensures that PSO correctly runs on all worker nodes in your cluster. It will keep the config updated and ensure that files are installed safely.

As previously mentioned, the installation of the Pure Service Orchestrator for Kubernetes requires that you have Helm3 installed on your Kubernetes cluster. After you have installed the Helm3 binaries and completed the installation, you should perform the following steps:

1. Add the pure repo to Helm:

```
# helm repo add pure http://purestorage.github.io/pso-csi
# helm repo update
# helm search repo pure-psos
```

2. Update the PSO configuration file: Enable Pure Service Orchestrator for Kubernetes to communicate with your Pure Storage backend arrays, by updating the PSO configuration file to reflect the access information for the backend storage solutions. The file is called values.yaml and needs to contain the management IP address of the backend devices, together with a valid, privileged, API token for each device. Additionally, an NFS Data VIP address is required for each FlashBlade.
3. Take a copy of the values.yaml provided by the Helm Chart⁵ and update the parameters for the arrays in the configuration file with your site-specific information, as shown in the following example:

⁵ Or download from <https://raw.githubusercontent.com/purestorage/pso-csi/master/pure-psos/values.yaml>



```

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"

```

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

Configure the parameter clusterID to be a unique value to identify your Kubernetes cluster. This ensures that multiple Kubernetes clusters running with PSO can coexist on the same backends without fear of volume and share name clashes. If you wish to use Fibre Channel as your data protocol for FlashArrays, then you must also change the following parameter in the configuration file:

```
flasharray.sanType: FC
```

Please note that Fibre Channel support is only for bare-metal installation.

4. Create a Namespace for PSO: Pure requires that PSO is installed into its own namespace, therefore create a namespace with the following command:

```
kubectl create namespace <name>
```

5. Install the plugin: It is advisable to perform a ‘dry run’ installation to ensure that your YAML file is correctly formatted:

```
# helm install pure-psos pure/pure-psos -f
<your_own_dir>/<your_own_values>.yaml --namespace <name> --dry-run --debug
```

Perform the actual install.

```
# helm install pure-psos pure/pure-psos -namespace <name> -f
<your_own_dir>/<your_own_values>.yaml
```

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

```
# helm install pure-psos pure/pure-psos -namespace <name> -f
<your_own_dir>/<your_own_values>.yaml --set flasharray.sanType=FC
```

The recommendation is to use the values.yaml file rather than the --set option for ease of use, especially should modifications be required to your configuration in the future.



Validate Correct Plugin Installation

To ensure that the Pure Service Orchestrator is correctly installed and running, we need to check for a few simple things in the Kubernetes cluster. Note there will also be CockroachDB related items in the namespace.

StorageClass: Two ‘pure’ classes should exist.

```
# kubectl get sc
```

NAME	PROVISIONER	AGE
pure-block	pure-provisioner	2d
pure-file	pure-provisioner	2d

DaemonSet:

```
# kubectl get ds -n <name>
```

NAME	DESIRED	CURRENT	READY	UP-TO-DATE	AVAILABLE	NODE SELECTOR	AGE
pso-csi-node	6	6	6	6	6	<none>	2d

Deployment:

```
# kubectl get statefulset -n <name>
```

NAME	READY	AGE
pso-csi-controller	1/1	2d

Service:

```
# kubectl get service -n <name>
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
pso-csi-controller	ClusterIP	10.233.34.208	<none>	12345/TCP	2d

Pods: One pso-csi-node pod should be running on each cluster node, one pso-csi-controller pod plus between 5 and 7 pso-db pods

```
# kubectl get pod --namespace <name>
```

NAME	READY	STATUS	RESTARTS	AGE
pso-csi-controller-0	6/6	Running	0	2d
pso-csi-node-dhpxf	3/3	Running	0	2d
pso-csi-node-jb8vn	3/3	Running	0	2d
pso-csi-node-k6q21	3/3	Running	0	2d
pso-csi-node-p9pwh	3/3	Running	0	2d
pso-csi-node-rndzj	3/3	Running	0	2d
pso-csi-node-w7fpg	3/3	Running	0	2d



Scaling Backend Storage

As your CaaS platform scales with increased demand from applications, workflows and users, you'll inevitably face a demand for additional backend persistent storage to support these applications and workflows.

You may have a block-only persistent storage environment and have been requested to add a file-based solution as well, or your current block and file backends may be reaching capacity limits. Additionally, you may want to add or change existing labels.

With the Pure Service Orchestrator, adding additional storage backends or changing labels is seamless and straightforward. The process is as simple as updating your configuration YAML file with new labels or adding new FlashArray or FlashBlade access information and then running this single command:

```
# helm upgrade pure-psos pure/pure-psos -f <your_own_dir>/<your_own_values>.yaml
```

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

Simple Pod Deployment with a Pure Storage FlashArray Persistent Volume

To validate that the Pure Storage Kubernetes plugin has been configured and installed correctly, we can create a simple pod with the running Microsoft® SQL Server® for Linux using a persistent volume from the configured Pure Storage backend. Provided here are two files that we can use to validate the installation and show a working application deployment. These are YAML files, firstly defining a Pure Storage-based persistent volume claim, and secondly defining the Nginx application using the persistent volume.

Before we can do any of this work, the SQL Server is going to require a password, and this is obtained from a Kubernetes secret. To create the secret, we must issue this command:

```
# kubectl create secret generic mssql --from-literal=SA_PASSWORD=<your password>
```

Persistent Volume Claim

Create a file called **sql-pvc.yaml**:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mssql-data
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
  storageClassName: pure-block
```



TECHNICAL WHITE PAPER

Execute the following command:

```
# kubectl create -f sql-pvc.yaml
```

This will create a PVC called mssql-data and the Pure Storage Dynamic Provisioner will automatically create a Persistent Volume to back this claim and be available to a pod that requests it.

The PV created for the PVC can be seen using the following command:

```
# kubectl get pvc
```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
mssql-data	Bound	pvc-41c347e9-968f-11e8-9c45-0025b5c0808f	8Gi	RWO	pure-block	18s

From this, we can cross-reference to the actual volume created on FlashArray.

The screenshot shows the Pure Storage FlashArray GUI interface. The left sidebar has navigation links for Dashboard, Storage, Analysis, Health, and Settings. The main area is titled 'Storage' and has tabs for Array, Hosts, Volumes, Protection Groups, and Pods. The 'Volumes' tab is active, indicated by an orange underline. Below the tabs, there's a breadcrumb navigation: Storage > Volumes > k8s-pvc-41c347e9-968f-11e8-9c45-0025b5c0808f. A summary table provides details: Size (12093683744 K), Data Reduction (8.9 to 1), Volumes (5.61 G), Snapshots (0.00), Shared (2.66 G), System (0.00), and Total (8.27 G). The main content area displays a list of volumes. One volume is selected: 'k8s-pvc-41c347e9-968f-11e8-9c45-0025b5c0808f'. Its details are shown in a table: Name (k8s-pvc-41c347e9-968f-11e8-9c45-0025b5c0808f), Size (8 GiB), Volumes (0.00), Snapshots (0.00), and Reduction (1.0 to 1). Below the table, it says 'Destroyed (0)'.

Figure 2. FlashArray GUI Storage > Volumes Pane

We can see that the volume name matches the PV name with a prefix of k8s-. This prefix is technically the **ClusterID** parameter defined in the `values.yaml` configuration file mentioned previously. Looking more closely at the volume on FlashArray, we see that it is also not yet connected to any host, as no pod is using the volume.

The screenshot shows the Pure Storage FlashArray GUI interface, similar to Figure 2 but for a specific volume. The left sidebar and top navigation are identical. The main area is titled 'Storage' and has tabs for Array, Hosts, Volumes, Protection Groups, and Pods. The 'Volumes' tab is active. The breadcrumb navigation shows Storage > Volumes > k8s-pvc-41c347e9-968f-11e8-9c45-0025b5c0808f. A summary table for this volume shows: Size (8 GiB), Data Reduction (1.0 to 1), Volumes (0.00), Snapshots (0.00), Shared (-), System (-), and Total (0.00). Below the summary, a section titled 'Connected Hosts' is present. It has a table with a single row: 'Name' (empty) and 'LUN' (empty). It also says '0 of 0' and 'No hosts found.'

Figure 3. FlashArray GUI Storage > Selected Volume Pane



Application Deployment

Create a file called **sqldeployment.yaml**:

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: mssql-deployment
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: mssql
    spec:
      terminationGracePeriodSeconds: 10
      containers:
        - name: mssql
          image: microsoft/mssql-server-linux
          ports:
            - containerPort: 1433
          securityContext:
            privileged: true
          env:
            - name: ACCEPT_EULA
              value: "Y"
            - name: SA_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: mssql
                  key: SA_PASSWORD
          volumeMounts:
            - name: mssqldb
              mountPath: /var/opt/mssql
      volumes:
        - name: mssqldb
          persistentVolumeClaim:
            claimName: mssql-data
```

Execute the following command:

```
# kubectl create -f sqldeployment.yaml
```



TECHNICAL WHITE PAPER

This will create a pod for the mssql-deployment running SQL Server for Linux and the Pure Service Orchestrator will mount the PV created earlier to the directory /var/opt/mssql within the pod. To find the exact name of the pod created use the ‘kubectl get pods’ command. A lot of information can be gathered regarding the newly created pod – some useful information is highlighted below:

```
# kubectl describe pod mssql-deployment-5f9b58fd9b-2nzfm

Name:mssql-deployment-5f9b58fd9b-2nzfm
Namespace:default
Node:sni-c08-caas-02/10.21.200.62
Start Time: Thu, 03 Sep 2020 14:25:42 -0700
Labels: app=mssql
Annotations: <none>
Status: Running
IP: 10.233.80.205
Controlled By: ReplicaSet/mssql-deployment-5f9b58fd9b
Containers:
mssql:
Container ID: docker://f38ea59cb674f83aa1db18508029021d131806da8fe0d423767d8ef3260e514c
Image: microsoft/mssql-server-linux
Image ID: docker-pullable://microsoft/mssql-server-
linux@sha256:8231b746946d12a6a1d5e6c7bcb3d983e97ed1fe5ba3ad04e52305060aced166
Port: 1433/TCP
Host Port: 0/TCP
State: Running
Started: Thu, 03 Sep 2020 14:25:46 -0700
Ready: True
Restart Count: 0
Environment:
ACCEPT_EULA: Y
SA_PASSWORD: <set to the key 'SA_PASSWORD' in secret 'mssql'> Optional: false
Mounts:
/var/opt/mssql from mssql (rw)
/var/run/secrets/kubernetes.io/serviceaccount from default-token-7v5xw (ro)
Conditions:
Type Status
Initialized True
Ready True
PodScheduled True
Volumes:
mssql:
Type: PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same
namespace)
ClaimName: mssql-data
ReadOnly: False
default-token-7v5xw:
Type: Secret (a volume populated by a Secret)
SecretName: default-token-7v5xw
Optional: False
```



TECHNICAL WHITE PAPER

```
QoS Class:           BestEffort
Node-Selectors:     <none>
Tolerations:       <none>
-
-
-
```

We can see which node the pod has been created on, and this can be confirmed from the FlashArray GUI.

Size	Data Reduction	Volumes	Snapshots	Shared	System	Total
8 G	31.5 to 1	121.26 K	0.00	-	-	121.26 K

Figure 4. FlashArray GUI Storage > Selected Volume Pane

We can confirm the SQL Server application is working by running sqlcmd to connect to the pod at its internal IP address, which we can find in the pod description above, and then a couple of simple SQL commands to prove the database is there.

```
# /opt/mssql-tools/bin/sqlcmd -S 10.233.80.205 -U sa -P <password>
1> select @@servername
2> go
```



```
-----  
mssql-deployment
```

```
(1 rows affected)
1> select name, database_id, create_date from sys.databases ;
2> go
```

Name	database_id	create_date
Master	1	2003-04-08 09:13:36.390
Tempdb	2	2020-09-03 21:25:51.700
Model	3	2003-04-08 09:13:36.390
Msdb	4	2020-06-30 00:03:38.280

```
(4 rows affected)
1>
```

Simple Multiple Pod Deployment with a Pure Storage FlashBlade Persistent Volume

Here we are going to validate that the Pure Service Orchestrator plugin has been configured and installed correctly to creates NFS based persistent volumes on a Pure Storage FlashBlade backend, that can be shared by multiple pods. Provided here are files that we can use to validate the installation and show an end-to-end example. These are YAML files firstly defining a Pure Storage based persistent volume claim, secondly defining the Nginx application using the persistent volume and finally defining an additional pod to connect to the same PVC.

Persistent Volume Claim

Create a file called **nginx-nfs-pvc.yaml**:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pure-nfs-claim
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
    storageClassName: pure-file
```

Execute the following command:

```
# kubectl create -f nginx-nfs-pvc.yaml
```



This will create a PVC called pure-nfs-claim and the Pure Service Orchestrator will automatically create a Persistent Volume to back this claim and be available to a pod that requests it.

The PV created for the PVC can be seen using the following command:

```
# kubectl get pvc
```

NAME	STATUS	VOLUME	CAPACITY	ACCESSMODES	STORAGECLASS	AGE
pure-nfs-claim	Bound	pvc-a59d3f5a-997a-11e8-9c45-0025b5c0808f	10Gi	RWX	pure-file	12s

and from this, we can cross-reference to the actual volume created on the Pure Storage FlashBlade.

Name	Size	Used	% Used	Data Reduction	Physical	Snapshots	Protocols
k8	10 G	0.00	0%	-	0.00	0.00	NFS

Figure 5. FlashBlade GUI Storage > Storage File Systems Pane

Again, we can see that the filesystem name matches the PV name with a prefix of k8s-.

Nginx Application Pod

Create a file called **nginx-pod-nfs.yaml**:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-nfs
  namespace: default
spec:
  volumes:
    - name: pure-nfs
      persistentVolumeClaim:
        claimName: pure-nfs-claim
  containers:
    - name: nginx-nfs
      image: nginx
      command:
        - sleep
        - "3600"
      volumeMounts:
        - name: pure-nfs
          mountPath: /data
  ports:
    - name: pure
      containerPort: 80
```

Execute the following command:

```
# kubectl create -f nginx-pod-nfs.yaml
```

This will create a pod called nginx-pod-nfs that will run the Nginx image and the CSI driver will mount the PV created earlier to the directory /data within the pod. A lot of information can be gathered regarding the newly created pod as shown below, but some useful information is highlighted below:



TECHNICAL WHITE PAPER

```
# kubectl describe pod nginx-nfs
Name:nginx-nfs
Namespace:default
Node:sn1-c08-caas-01/10.21.200.61
Start Time: Mon, 07 Sep 2020 06:22:00 -0700
Labels:<none>
Annotations: <none>
Status:Running
IP: 10.233.90.13
Containers:
  nginx-nfs:
    Container ID: docker://420ae7b6172425432dfa8faacd0b583a4f43235ba622936b9a235e61abe29837
    Image: nginx
    Image ID: docker-
    pullable://nginx@sha256:d85914d547a6c92faa39ce7058bd7529baacab7e0cd4255442b04577c4d1f424
    Port: 80/TCP
    Host Port: 0/TCP
    State: Running
    Started: Mon, 07 Sep 2020 06:22:08 -0700
    Ready: True
    Restart Count: 0
    Environment: <none>
    Mounts:
      /data from pure-nfs (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-7v5xw (ro)
Conditions:
  Type Status
  Initialized True
  Ready True
  PodScheduled True
Volumes:
  pure-nfs:
    Type: PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same
          namespace)
    ClaimName: pure-nfs-claim
    ReadOnly: false
  default-token-7v5xw:
    Type: Secret (a volume populated by a Secret)
    SecretName: default-token-7v5xw
    Optional: false
  QoS Class: BestEffort
  Node-Selectors: <none>
```



Additional Application Pod

Create a new pod definition file called **busybox-nfs.yaml**:

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox-nfs
  namespace: default
spec:
  volumes:
    - name: pure-nfs-2
      persistentVolumeClaim:
        claimName: pure-nfs-claim
  containers:
    - name: busybox-nfs
      image:
        busybox
      volumeMounts:
        - name: pure-nfs-2
          mountPath: /usr/share/busybox
```

Execute the following command:

```
# kubectl create -f busybox-nfs.yaml
```

This will create a second pod, running in the same namespace as the Nginx pod, however we are using the same backing store by using the same claim name. A lot of information can be gathered regarding the newly created pod as shown below, but some useful information is highlighted below:

```
# kubectl describe pod busybox-nfs

Name:busybox-nfs
Namespace:default

Node: sn1-c08-caas-08/10.21.200.68
Start Time: Mon, 07 Sep 2020 06:27:13 -0700
Labels: <none>
Annotations: <none>
Status: Running
IP: 10.233.121.142
Containers:
  busybox-nfs:
    Container ID: docker://92bbe9949f7ba6251a9035fc33567a2067e3fa422d01014747402d66f9628655
    Image: busybox
```



TECHNICAL WHITE PAPER

```
Image ID: docker-
pullable://busybox@sha256:cb63aa0641a885f54de20f61d152187419e8f6b159ed11a251a09d115fdff9bd
  Port: <none>
  Host Port: <none>
  State: Waiting
  Reason: CrashLoopBackOff
Last State: Terminated
  Reason: Completed
  Exit Code: 0
  Started: Mon, 07 Sep 2020 06:27:40 -0700
  Finished: Mon, 07 Sep 2020 06:27:40 -0700
  Ready: False
  Restart Count: 2
  Environment: <none>
Mounts:
  /usr/share/busybox from pure-nfs-2 (rw)
  /var/run/secrets/kubernetes.io/serviceaccount from default-token-7v5xw (ro)
Conditions:
Type          Status
Initialized   True
Ready         False
PodScheduled  True
Volumes:
pure-nfs-2:
  Type:      PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same
             namespace)
  ClaimName: pure-nfs-claim
  ReadOnly:  false
default-token-7v5xw:
  Type:      Secret (a volume populated by a Secret)
  SecretName: default-token-7v5xw
  Optional:  false
  QoS Class: BestEffort
  Node-Selectors: <none>
```

It can be seen that both the nginx and busybox pods are using the same storage claim that is attached to the same NFS mount point on the backend, but each pod is actually running on a different node in the Kubernetes cluster.



Adding Nodes to a Kubernetes Cluster

This reference implementation is large enough to provide enough resources to run a few simple applications (see Appendix). However, as a cluster becomes more utilized, it may be necessary to provide additional application or infrastructure nodes to support other resource requirements.

There are processes for adding additional nodes to a Kubernetes cluster, and they are well documented within the main Kubernetes documentation set. Still, it is essential to cover how to ensure that additional nodes have the ability to utilize the Pure Storage arrays as providers of stateful storage.

When it comes to ensuring your new node can access stateful storage on Pure Storage devices, it is good to note that, as we are using a DaemonSet to ensure that our plugin is correctly installed on cluster nodes, the addition of a new cluster node to your Kubernetes cluster will cause the DaemonSet to create a new pure-csi pod on the new node and install the plugin correctly.

Conclusion

With the growth of applications and deployments that require a CaaS platform that can also provide an underlying stateful storage solution, the Pure Storage Kubernetes plugin meets these needs.

Additionally, using Pure Storage products to provide stateful storage also enables storage that is enterprise-ready, redundant, fast, resilient, and scalable.

Appendix: Application Examples

Here we show two simple application deployments that can be used in a Kubernetes environment that would also require stateful storage.

MongoDB

Running MongoDB in a HA configuration is a good example of how, using pre-existing Helm charts, you can easily deploy a production ready application seamlessly using Kubernetes StatefulSets with Pure Service Orchestrator providing the persistent storage from a Pure Storage FlashArray.

Here we are going to use a ‘stable’ Helm chart to create a MongoDB deployment using ReplicaSets. The database will be deployed with a primary and two secondary pods, each having their own persistent volume. To show that MongoDB replication is working we will add some data into the database on the primary and then read it from one of the secondaries. Notice that we only need to supply the name of the storageClass to the Helm configuration because the statefulSet comes with a template for creating new PVCs as the deployment scales.



TECHNICAL WHITE PAPER

```
# kubectl create namespace caas-mongo
# helm install --set "auth.adminUser=admin, auth.adminPassword=password,
persistentVolume.storageClass=pure" stable/mongodb-replicaset
NAME: caas-mongo
LAST DEPLOYED: Mon Sep 7 07:37:31 2020
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/ConfigMap
NAME                      DATA   AGE
caas-mongo-mongodb-replicaset-init 1      0s
caas-mongo-mongodb-replicaset-mongodb 1      0s
caas-mongo-mongodb-replicaset-tests 1      0s

==> v1/Service
NAME           TYPE      CLUSTER-IP    EXTERNAL-IP   PORT(S)      AGE
caas-mongo-mongodb-replicaset   ClusterIP  None         <none>        27017/TCP   0s

==> v1beta2/StatefulSet
NAME          DESIRED  CURRENT  AGE
caas-mongo-mongodb-replicaset  3        1        0s

==> v1/Pod(related)
NAME          READY  STATUS     RESTARTS  AGE
caas-mongo-mongodb-replicaset-0  0/1    Init:0/3  0          0s

# kubectl get pvc
NAME          STATUS  VOLUME                                     CAPACITY
ACCESS MODES  STORAGECLASS
datadir-caas-mongo-mongodb-replicaset-0  Bound  pvc-3ff67917-9986-11e8-9b47-0025b5c0807f  10Gi
RWO          pure
47m
datadir-caas-mongo-mongodb-replicaset-1  Bound  pvc-5ca21b0f-9986-11e8-9b47-0025b5c0807f  10Gi
RWO          pure
46m
datadir-caas-mongo-mongodb-replicaset-2  Bound  pvc-770a92f6-9986-11e8-9b47-0025b5c0807f  10Gi
RWO          pure
45m
```

Stateful Sets					
Name	Labels	Pods	Age	Images	
caas-mongo-mongodb-replicaset	app: mongodb-replicaset chart: mongodb-replicaset-3.5.2 heritage: Tiller release: caas-mongo	3 / 3	53 minutes	mongo:3.6 busybox k8s.gcr.io/mongodb:install:0.6 mongo:3.6	✓ ⋮

Figure 6. MongoDB Stateful Sets



The screenshot shows the Kubernetes dashboard under the 'Config and storage' section, specifically the 'Persistent Volume Claims' page. On the left, there's a sidebar with navigation links like Cluster, Namespaces, Nodes, Persistent Volumes, Roles, Storage Classes, Namespace (set to default), and Overview. The main area has a header 'Persistent Volume Claims' with columns: Name, Status, Volume, Capacity, Access Modes, Storage Class, and Age. Three entries are listed, all labeled 'Bound' and have a 'ReadWriteOnce' access mode and 'pure' storage class. The volumes are named pvc-770a92f6-9986-11e8-9b47-0025b5c0807f, pvc-5ca21b0f-9986-11e8-9b47-0025b5c0807f, and pvc-3f67917-9986-11e8-9b47-0025b5c0807f. The ages range from 53 minutes to 54 minutes.

Figure 7. MongoDB Persistent Volume Claims

Here we'll access the Primary server for the MongoDB deployment and add some simple entries into the database.

```
# kubectl exec -it caas-mongo-mongodb-replicaset-0 -- mongo --host caas-mongo-mongodb-replicaset
MongoDB shell version v3.6.6
connecting to: mongodb://caas-mongo-mongodb-replicaset:27017/
MongoDB server version: 3.6.6
Welcome to the MongoDB shell.

rs0:PRIMARY> db.products.insert({manufactirer:'Pure Storage',
product:'FlashArray'}) WriteResult({ "nInserted" : 1 })
rs0:PRIMARY> db.products.insert({manufactirer:'Pure Storage',
product:'FlashBlade'}) WriteResult({ "nInserted" : 1 })
rs0:PRIMARY> db.products.find()
{ "_id" : ObjectId("5b68602213827b834aa4a62e"), "manufactirer" : "Pure Storage", "product" : "FlashArray" }
{ "_id" : ObjectId("5b68602a13827b834aa4a62f"), "manufactirer" : "Pure Storage", "product" : "FlashBlade" }
} rs0:PRIMARY> exit
```

Now, let's interrogate one of the secondary nodes to ensure the data has been correctly replicated.

```
# kubectl exec -it caas-mongo-mongodb-replicaset-1 -- mongo --
eval="rs.slaveOk(); db.products.find().forEach(printjson)"
MongoDB shell version v3.6.6
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.6.6
{
    "_id" : ObjectId("5b68602213827b834aa4a62e"),
    "manufactirer" : "Pure Storage",
    "product" : "FlashArray"
}

{
    "_id" : ObjectId("5b68602a13827b834aa4a62f"),
    "manufactirer" : "Pure Storage",
    "product" : "FlashBlade"
}
```



WordPress

In this example we are using another ‘stable’ Helm chart to deploy WordPress, the content management system. The chart will be used to deploy a production-ready configuration with three WordPress pods, as well as a MariaDB deployment for the database requirements of the WordPress application.

The MariaDB deployment will use a persistent volume from a FlashArray and the WordPress pods will all use the same ReadWriteMany persistent volume made available from a FlashBlade.

The production-values.yaml was copied from the Helm chart github and the following simple modifications were made to ensure the required storage comes from the FlashArray and FlashBlade:

```
# kubectl create namespace caas-wordpress
# helm install -f ./wordpress-production.yaml
stable/wordpress NAME: caas-wordpress
LAST DEPLOYED: Mon Sep  7 08:50:07 2020
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1beta1/Deployment
NAME          DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
caas-wordpress-wordpress  3        3        3           0          1s

==> v1beta1/StatefulSet
NAME          DESIRED  CURRENT  AGE
caas-wordpress-mariadb  1        1        1s

==> v1beta1/Ingress
NAME          HOSTS          ADDRESS      PORTS  AGE
wordpress.local-caas-wordpress  wordpress.local  80, 443  1s

==> v1/Pod(related)
NAME          READY  STATUS      RESTARTS  AGE
caas-wordpress-wordpress-5b45fc89c5-dtz8q  0/1  ContainerCreating  0          1s
caas-wordpress-wordpress-5b45fc89c5-f52xq  0/1  ContainerCreating  0          1s
caas-wordpress-wordpress-5b45fc89c5-ntds9  0/1  ContainerCreating  0          1s
caas-wordpress-mariadb-0                   0/1  ContainerCreating  0          1s

==> v1/Secret
NAME          TYPE  DATA  AGE
caas-wordpress-mariadb  Opaque  2    1s
caas-wordpress-wordpress  Opaque  2    1s

==> v1/ConfigMap
NAME          DATA  AGE
caas-wordpress-mariadb  1    1s
caas-wordpress-mariadb-tests  1    1s
```



TECHNICAL WHITE PAPER

```
==> v1/PersistentVolumeClaim
NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE
caas-wordpress-wordpress Pending pure-file 1s

==> v1/Service
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
caas-wordpress-mariadb ClusterIP 10.233.9.205 <none> 3306/TCP 1s
caas-wordpress-wordpress ClusterIP 10.233.35.83 <none> 80/TCP, 443/TCP 1s
```

Looking at the Kubernetes dashboard we can see the persistent volumes associated with the deployment.

Name	Status	Volume	Capacity	Access Modes	Storage Class	Age
caas-wordpress-wordpress	Bound	pvc-648846ef-9990-11e8-9b47-0025b5c0807f	10Gi	ReadWriteMany	pure-file	16 minutes
data-caas-wordpress-mariadb	Bound	pvc-64a30ae2-9990-11e8-9b47-0025b5c0807f	8Gi	ReadWriteOnce	pure-block	16 minutes

And also the four pods that have been created:

Name	Node	Status	Restarts	Age
caas-wordpress-wordpress-5b45fc89c5-ntds9	sn1-c08-caas-07	Running	0	17 minutes
caas-wordpress-mariadb-0	sn1-c08-caas-02	Running	0	17 minutes
caas-wordpress-wordpress-5b45fc89c5-dtz8q	sn1-c08-caas-06	Running	0	17 minutes
caas-wordpress-wordpress-5b45fc89c5-f52xq	sn1-c08-caas-05	Running	0	17 minutes

If we examine the MariaDB pod, we will see it is using the persistent volume from the FlashBlade:

Name	Status	Volume	Capacity	Access Modes	Storage Class	Age
data-caas-wordpress-mariadb	Bound	pvc-64a30ae2-9990-11e8-9b47-0025b5c0807f	8Gi	ReadWriteOnce	pure-block	19 minutes

And looking at one of the WordPress pods we can see it is using the ReadWriteMany volume from the FlashBlade:

Name	Status	Volume	Capacity	Access Modes	Storage Class	Age
caas-wordpress-wordpress	Bound	pvc-648846ef-9990-11e8-9b47-0025b5c0807f	10Gi	ReadWriteMany	pure-file	21 minutes



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.

[Read Simon's blog](#)

©2020 Pure Storage, the Pure P Logo, and the marks on the Pure Trademark List at <https://www.purestorage.com/legal/productenduserinfo.html> are trademarks of Pure Storage, Inc. Other names are trademarks of their respective owners. Use of Pure Storage Products and Programs are covered by End User Agreements, IP, and other terms, available at: <https://www.purestorage.com/legal/productenduserinfo.html> and <https://www.purestorage.com/patents>

The Pure Storage products and programs described in this documentation are distributed under a license agreement restricting the use, copying, distribution, and decompilation/reverse engineering of the products. No part of this documentation may be reproduced in any form by any means without prior written authorization from Pure Storage, Inc. and its licensors, if any. Pure Storage may make improvements and/or changes in the Pure Storage products and/or the programs described in this documentation at any time without notice.

THIS DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID. PURE STORAGE SHALL NOT BE LIABLE FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS DOCUMENTATION. THE INFORMATION CONTAINED IN THIS DOCUMENTATION IS SUBJECT TO CHANGE WITHOUT NOTICE.

Pure Storage, Inc.
650 Castro Street, #400
Mountain View, CA 94041

purestorage.com

800.379.PURE

