REFERENCE ARCHITECTURE

PLATFORM-AS-A-SERVICE

RED HAT OPENSHIFT 3.11 AND PURE STORAGE

TABLE OF CONTENTS

INTRODUCTION

This document provides a practical reference architecture to help integrate Pure Storage® products into the deployment of a Red Hat® OpenShift® Container Platform. The underlying infrastructure for this platform will be based on a bare-metal deployment that can be scaled easily to whatever size is required. It is assumed that the reader understands how to deploy a bare-metal OpenShift solution, as details will only be provided for the Pure Storage integration pieces. Links to details on OpenShift deployments can be found in the Appendix of this document.

PLATFORM-AS-A-SERVICE

PaaS Defined

Cloud computing has widened its scope to include platforms for developing and implementing custom applications, a term called "Platform as a Service" (PaaS). PaaS applications are also suggested as on-demand, web-based, or Software-as-a-Service (SaaS) options. However, a comprehensive definition is:

> *Platform as a Service (PaaS) is the delivery of a computing platform and solution stack as a service. PaaS offerings facilitate deployment of applications without the cost and complexity of buying and managing the underlying hardware and software and provisioning hosting capabilities, providing all of the facilities required to support the complete life cycle of building and delivering web applications and services entirely available on the Internet.*
>
> *PaaS offerings may include facilities for application design, application development, testing, deployment, and hosting. This includes the scope of application services such as team collaboration, web service integration and marshalling, database integration, security, scalability, storage, and developer community facilitation, among others. These services may be provisioned as an integrated solution offering over the web.*

In simple terms, PaaS provides a runtime environment for cloud applications. It refers to the almost negligible need to buy standalone software, hardware, and all related services, since these are available on the Internet in a more "public cloud" manner. In a private cloud setting, one would still need to buy hardware and software to build the infrastructure, but PaaS will help manage and utilize it in a manner that meets cloud standards. In the next section, we will discuss the cloud standards that any PaaS platform must implement. Additionally, in private cloud scenarios, these services might be available through an Intranet or other means.

After looking at the PaaS platforms available today, it appears that most true PaaS platforms provide a runtime environment for applications developed for a cloud. However, some PaaS providers target the development environment and provide an entire solution stack that can be used to build, test, deploy, and manage code in the cloud. Those providing development and testing services look more like SaaS – offering development or testing tools in the cloud. Although this is still a topic of debate, for the purposes of this paper we consider PaaS to be a runtime environment for cloud applications, and we will discuss the requirements and architecture of a PaaS platform in that context.

Requirements of a PaaS Platform

As stated above, the main objective of PaaS is to improve the efficiency of the cloud and maximize its benefits. Keeping this objective in mind, below are the requirements of an ideal PaaS platform:

- **High Scalability and On-Demand Provisioning** Infrastructure-as-a-Service (IaaS) provides scaling and on-demand hardware provisioning. In terms of cloud, this scaling is possible until the last hardware resource is available in cloud. Likewise, PaaS is expected to scale applications across the hardware, and the extent of scaling can be stretched to include the last hardware resource available for deployment. This provides users of PaaS a feeling of infinite scalability. In addition, the application provisioning should be an automated task that needs no IT intervention for deployment and delivery.
- **High Availability** PaaS platforms should provide a runtime environment for applications that features failover and load balancing capabilities. The important question is "how is it different from a traditional clustered, load-balanced environment?" The answer is that failover and load balancing capabilities should be scoped across the cloud rather than a few dedicated machines, as is the case in a traditional environment. This is over and above the hardware availability provided by IaaS. Thus, by deploying a PaaS platform, application availability is guaranteed in the event of application runtime breakdown and not infrastructure breakdown.
- **High Reliability** Reliability is often used interchangeably with availability. Though the motive of both is to provide a failover, there is a fine line that distinguishes one from the other. This difference can be made clear by means of an example: In the case of a business service that calculates an individual's federal and state taxes, let's first assume it is deployed in a cloud which provides only availability. In this scenario, whenever there is a request for a tax calculation, the cloud will ensure that some service is always up and running to receive this request. However, other processes running on the same computing environment could cause the service to take a long time to respond and the request to time out. In this case, the request initiator would see an error page. Now, had the cloud been reliable, it would have sensed that the service was not responding within the specified time and would have tried to execute it in another computing environment. In this case, the user would have received a response and not an error. A successful PaaS platform should provide this reliability to all services/components deployed and running on it.
- **Optimal Usage** One of the core requirements of any cloud computing platform is optimal usage of resources. In the case of PaaS, optimization specifically applies to resources utilized for executing applications. To apply resource optimization, the PaaS platform should have components that monitor application execution and usage. Another purpose of monitoring is to provide chargeback to users. Let us see how this requirement differs from its applicability to a traditional deployment. In traditional deployments, applications are load balanced using traditional hardware and software load balancers that monitor a few

application servers and distribute the load using various load balancing strategies such as "round robin" or "least recently used." In the PaaS context, since PaaS monitors the runtime for the individual services of an application, load balancing should be more granular. Here PaaS should monitor each service/component within the application based on different parameters (number of requests being serviced, CPU usage of the VM running the machine, etc.) and then decide on the best candidate to service the incoming request. PaaS is spread across the cloud, so this load balancing should not be limited to a few machines but to the entire cloud where the PaaS exists. The other optimization scenario where PaaS distinguishes itself from a traditional deployment is that of a service orchestration. Wherever services are executed in a workflow or process-based manner, PaaS should keep track of the current state of the workflow or process to ensure that work completed during execution of a process is not wasted if the process fails – rather than starting the process all over. This has the potential to salvage the computing loss due to failure and improve the efficiency of the cloud.

- **Auto-Scaling** On-demand scaling could be based on a user request or in response to an increased load. In the latter scenario, the cloud, because of its elastic nature, expands and adds more resources to meet the increased demand. This requires the PaaS to auto-scale the applications in the newly added computing resources.
- **Admin/Management Console and Reports** PaaS platforms should include some form of a user interface through which all application components/services can be tracked and monitored. In the case of private cloud, this UI may be integrated with the IaaS monitoring/tracking tool. In addition, this UI should have a provision for requesting additional deployments of applications/services along with access control for the same. PaaS platforms should also have reporting capabilities to provide statistics related to application usage, execution, and provisioning. If reporting capabilities are not present in the form of UI, then there should at least be APIs or web service interfaces that users of PaaS can use to build their own reports.
- **Multi-OS and Multi-Language Support** An organization may have different operating system and applications written using different languages. PaaS platforms should enable applications which can run on multiple operating systems (Windows, Linux, etc.) and should be able to run applications created in different languages (Java, .Net, C++, etc.).

PaaS – Bare Bones

The requirements discussed in the above section comprise both essential and useful-to-have features. Organizations can choose to have a partial implementation of these features to meet their PaaS requirements, because each organization may have varying needs with respect to scaling, availability, and reliability. The following are basic requirements of a homegrown PaaS platform, along with a discussion as to what extent of implementation is needed:

- **High Scalability & On-Demand Provisioning** This is one of the most basic requirements of PaaS for implementation. However, the scope of scalability could be adjusted to suit the application need under the cloud. Provisioning of applications has to be on-demand and without human intervention. Without implementing these two aspects, deploying PaaS would become futile.

- **High Availability** This requirement is also imperative, but, depending on the organization's needs, one could end up with a low failure threshold. Therefore, if the custom PaaS components that provide availability are finite, and if they all fail, there is a possibility that the PaaS will fail to accept a request.
- **High Reliability** This requirement can also be exposed to finite points of failure rather than infinite controllers providing infinite (scope entire cloud) reliability.
- **Optimal Usage** This requirement could be confined to load balancing to give the cloud advantage, but it must be granular and should be able to load balance individual services rather than the runtimes that these services run on.
- **Self-Service Portal** Instead of a full-fledged dashboard, one could deliver a simple portal that provides a UI to request cloud resources, including applications/services deployed in the PaaS. The rest of the prerequisites may or may not be implemented in a custom PaaS and would depend on the specific needs of the user organization.

COMPONENTS, PRE-REQUISITES, AND CONFIGURATION

Pure Storage FlashArray

The Pure Storage FlashArray family delivers purpose-built, software-defined all-flash power and reliability for businesses of every size. FlashArray is all-flash enterprise storage that is up to 10X faster, more space and power efficient, more reliable, and far simpler than other available solutions. Critically, FlashArray also costs less, with a TCO that's typically 50% lower than traditional performance disk arrays.

FlashArray//X is the first mainstream, 100% NVMe, enterprise-class all-flash array. //X represents a higher performance tier for mission-critical databases, top-of-rack flash deployments, and Tier 1 application consolidation. //X, at up to 3PB in 6U, with hundred-microsecond range latency and GBs of bandwidth, delivers an unprecedented level of performance density that makes possible previously unattainable levels of consolidation.

FlashArray//X is ideal for cost-effective consolidation of everything on flash. Whether accelerating a single database, scaling virtual desktop environments, or powering an all-flash cloud, there is an //X model that fits your needs.

PURITY FOR FLASHARRAY (PURITY//FA 5)

At the heart of every FlashArray is Purity Operating Environment software. Purity//FA5 implements advanced data reduction, storage management, and flash management features, enabling organizations to enjoy Tier 1 data services for all workloads, proven 99.9999% availability (inclusive of maintenance and generational upgrades), completely non-disruptive operations, 2X better data reduction versus alternative all-flash solutions, and the power and efficiency of DirectFlash™. Moreover, Purity includes enterprise-grade data security, comprehensive data protection options, and complete business continuity via ActiveCluster multi-site stretch cluster. All these features are included with every array.

FLASHARRAY SPECIFICATIONS

TECHNICAL SPECIFICATIONS*

	CAPACITY	PHYSICAL
//X10	Up to 55 TB / 53.5 TiB effective capacity** Up to 20 TB / 18.6 TiB raw capacity	3U 490 – 600 Watts (nominal – peak) 95 lbs (43.1 kg) fully loaded 5.12” x 18.94” x 29.72” chassis
//X20	Up to 275 TB / 251.8 TiB effective capacity** Up to 87 TB / 80.3 TiB raw capacity††	3U 620 – 688 Watts (nominal – peak) 95 lbs (43.1 kg) fully loaded 5.12” x 18.94” x 29.72” chassis
//X50	Up to 650 TB / 602.8 TiB effective capacity** Up to 183 TB / 171 TiB raw capacity†	3U 620 – 760 Watts (nominal – peak) 95 lbs (43.1 kg) fully loaded 5.12” x 18.94” x 29.72” chassis
//X70	Up to 1.3 PB / 1238.5 TiB effective capacity** Up to 366 TB / 320.1 TiB raw capacity†	3U 915 – 1345 Watts (nominal – peak) 97 lbs (44.0 kg) fully loaded 5.12” x 18.94” x 29.72” chassis
//X90	Up to 3 PB / 3003.1 TiB effective capacity** Up to 878 TB / 768.3 TiB raw capacity†	3U – 6U 1100 – 1570 Watts (nominal – peak) 97 lbs (44 kg) fully loaded 5.12” x 18.94” x 29.72” chassis
DIRECT FLASH SHELF	Up to 1.9 PB effective capacity** Up to 512 TB / 448.2 TiB raw capacity	3U 460 - 500 Watts (nominal – peak) 87.7 lbs (39.8kg) fully loaded 5.12” x 18.94” x 29.72” chassis

//X CONNECTIVITY

Onboard Ports (per controller)

- 2 x 1/10/25 Gb Ethernet
- 2 x 1/10/25 Gb Ethernet Replication
- 2 x 1Gb Management Ports

Host I/O Cards (3 slots/controller)

- 2-port 10GBase-T Ethernet
- 2-port 1/10/25 Gb Ethernet
- 2-port 40 Gb Ethernet
- 2 Port 50Gb Ethernet (NVMe-oF Ready)***
- 2-port 16/32 Gb Fibre Channel (NVMe-oF Ready)
- 4-port 16/32 Gb Fibre Channel (NVMe-oF Ready)

* Stated //X specifications are applicable to //X R2 versions, expected availability June, 2018.

** 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.

*** Expected Availability 2H 2018.

† Array accepts Pure Storage DirectFlash Shelf and/or Pure Storage SAS-based expansion shelf.

†† Array accepts Pure Storage SAS-based expansion shelf.

Pure Storage FlashBlade

FlashBlade™ is a new, innovative scale-out storage system designed to accelerate modern analytics applications while providing best-of-breed performance in all dimensions of concurrency – including IOPS, throughput, latency, and capacity. FlashBlade is as simple as it is powerful, offering elastic scale-out storage services at every layer alongside DirectFlash™ technology for global flash management.

PURPOSE-BUILT FOR MODERN ANALYTICS

FlashBlade is the industry's first cloud-era flash purpose-built for modern analytics, delivering unprecedented performance for big data applications. Its massively distributed architecture enables consistent performance for all analytics applications using NFS, S3/Object, SMB, and HTTP protocols.

FAST

- Elastic performance that grows with data, up to 17 GB/s
- Always-fast, from small to large files
- Massively parallel architecture from software to flash

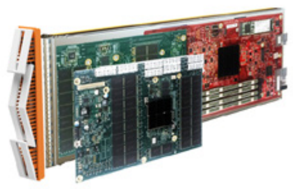
BIG

- Petabytes of capacity
- Elastic concurrency, up to 10s of thousands of clients
- 10s of billions of objects and files

SIMPLE

- Evergreen™ – don't rebuy TBs you already own
- "Tuned for Everything" design, no manual optimizations required
- Scale-out everything instantly by simply adding blades

THE FLASHBLADE DIFFERENCE



BLADE

Compute and network integrated with DirectFlash technology – each blade can be hot-plugged into the system for expansion and performance

PURITY//FB

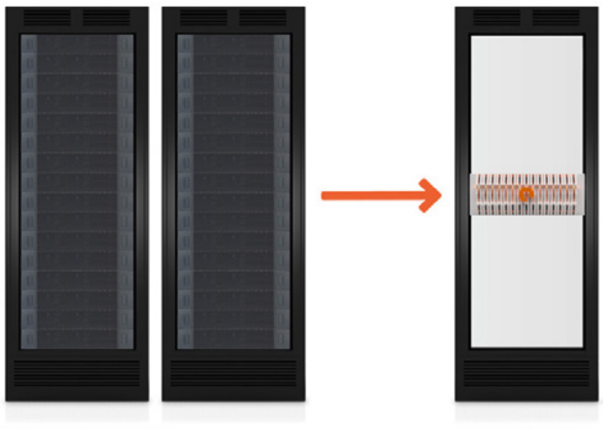
The heart of FlashBlade, architected on a massively distributed key-value pair database for limitless scale and performance, delivering enterprise-class data services and management with simplicity.

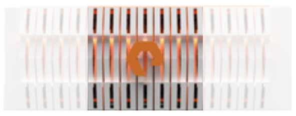
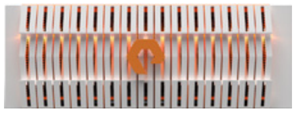
ELASTIC FABRIC

Powered by a proprietary object messaging protocol for fastest communication to flash, the low-latency converged fabric delivers a total bandwidth of 320Gb/s per chassis with 8x 40GB/s ports.

POWER, DENSITY, EFFICIENCY

FlashBlade delivers industry-leading throughput, IOPS, latency, and capacity – with up to 20x less space and 10x less power and cooling.



FLASHBLADE SPECIFICATIONS		8 TB BLADE	17 TB BLADE	52 TB BLADE
7 BLADES		98 TBs Usable	197 TBs Usable	591 TBs Usable
15 BLADES		267 TBs Usable	535 TBs Usable	1607 TBs Usable

* Usable capacity assumes 3:1 data reduction rate. Actual data reduction may vary based on use case.

PERFORMANCE

17 GB/s bandwidth with 15 blades

Up to 1.8M NFS ops/sec

CONNECTIVITY

8x 40Gb/s or 32x 10Gb/s Ethernet ports / chassis

PHYSICAL

4U

1,800 Watts (nominal at full configuration)

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 ultra-fast performance at scale. It also supports File protocol, 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 login. Pure1 Manage is SaaS-based, allowing you to manage your array from any browser or from 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 true performance forecasting – giving customers complete visibility into the performance and capacity needs of their 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 unrivaled 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.

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.

Red Hat OpenShift for Containers

Red Hat OpenShift is a layered system designed to expose an underlying Docker-formatted container image and Kubernetes concepts as accurately as possible, with a focus on easy composition of applications by a developer.

OpenShift Container Platform has a microservices-based architecture of smaller, decoupled units that work together. It runs on top of a Kubernetes cluster, with data about the objects stored in **etcd**, a reliable, clustered key-value store. Those services are broken down by function:

- **REST APIs**, which expose each of the core objects, such as projects, users, pods, services, images, etc.
- **Controllers**, which read those APIs, apply changes to other objects, and report status or write back to the object.

Users make calls to the REST API to change the state of the system. Controllers use the REST API to read the user's desired state, and then try to bring the other parts of the system into sync. For example, when a user requests a build, they create a "build" object. The build controller sees that a new build has been created, and runs a process on the cluster to perform that build. When the build completes, the controller updates the build object via the REST API, and the user sees that their build is complete.

The controller pattern means that much of the functionality in the OpenShift Container Platform is extensible. The way that builds are run and launched can be customized independently of how images are managed, or how deployments happen. The controllers are performing the "business logic" of the system, taking user actions and transforming them into reality. By customizing those controllers or replacing them with your own logic, different behaviours can be implemented. From a system administration perspective, this also means the API can be used to script common administrative actions on a repeating schedule. Those scripts are also controllers that watch for changes and act on these changes accordingly. The OpenShift Container Platform makes the ability to customize the cluster in this way a first-class behaviour.

To make this possible, controllers leverage a reliable stream of changes to the system to sync their view of the system with what users are doing. This event stream pushes changes from etcd to the REST API and then to the controllers as soon as changes occur, so changes can ripple out through the system very quickly and efficiently. However, since failures can occur at any time, the controllers must also be able to get the latest state of the system at startup and confirm that everything is in the right state. This resynchronization is important because it means that even if something goes wrong, the operator can restart the affected components, and the system double checks everything before continuing. The system should eventually converge to the user's intent since the controllers can always bring the system into sync.

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 moves 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, and 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 your developers an experience similar to what 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:** Supports both file and block
- **Enterprise-grade:** Delivers the same Tier 1 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 next generation, containerized data centers 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.

High-Level Design

The reference architecture used and described in this document, considered to be the minimum for a production level environment, consists of a single bastion host (also referred to as the Ansible control host), three master hosts, and five node hosts to run the actual Docker containers for the users. The master nodes run the clustered etcd key-value store. The node hosts are separated into two classes: infrastructure nodes and app nodes. The infrastructure nodes run the internal OpenShift Container Platform services, the OpenShift router, and the Local Registry. The remaining three app nodes host the actual user container processes. There is also a node which runs HAProxy to control access to the different functions of the OpenShift cluster.

This is shown in the figure on the next page.

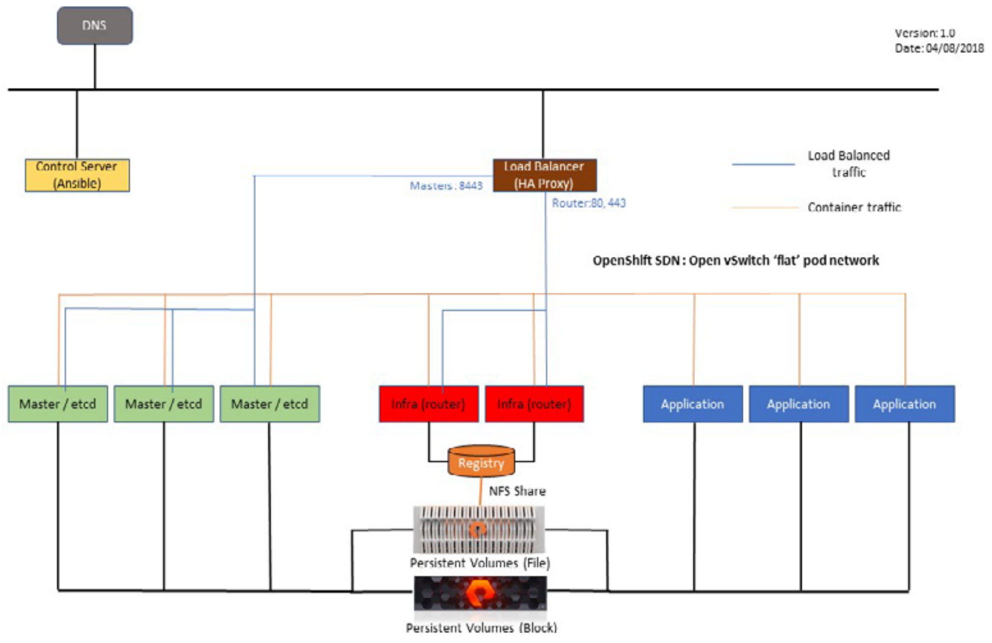


FIGURE 1. High-level architectural design

Within this reference architecture, we will employ Pure Storage products for several different uses. Specifically, these will be:

- The Docker service storage on each bare-metal node will be created on a block storage volume presented from Pure Storage FlashArray.
- The Docker Registry will be created within an NFS share presented from Pure Storage FlashBlade.
- Persistent Storage Volumes (PV) will be made available through the Pure Service Orchestrator plugin to allow PV claims to be made on block storage from Pure Storage FlashArray and from NFS mount points presented from Pure Storage FlashBlade.

Software Version Details

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

SOFTWARE	VERSION
RED HAT ENTERPRISE LINUX 7.5 X86_64	KERNEL-3.10.0-957
OPENSHIFT CONTAINER PLATFORM	3.11.92
KUBERNETES	1.11

DOCKER	1.13.1
ANSIBLE	2.6.4
PURE SERVICE ORCHESTRATOR	2.3.0

A subscription to the following Red Hat channels is also required to deploy this reference architecture.

CHANNEL	REPOSITORY NAME
RED HAT ENTERPRISE LINUX 7 SERVER (RPMS)	RHEL-7-SERVER-RPMS
RED HAT OPENSHIFT CONTAINER PLATFORM 3.11 (RPMS)	RHEL-7-SERVER-OSE-3.11-RPMS
RED HAT ENTERPRISE LINUX 7 SERVER – EXTRAS (RPMS)	RHEL-7-SERVER-EXTRAS-RPMS
RED HAT ANSIBLE ENGINE 2.6 RPMS FOR RED HAT ENTERPRISE LINUX 7	RHEL-7-SERVER-ANSIBLE-2.6-RPMS

Compute

The role of this Reference Architecture is not to prescribe specific compute platforms for the OpenShift platform, therefore we are referring to the servers being used in white-box terms. The servers used here have the following hardware specifications:

- Intel® Xeon® E5-2640 v2 @ 2.00GHz
- 32 vCPU
- 384 GiB memory

Networking

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

- 1 x 10GbE (management)
- 1 x dual Mellanox 40GbE (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 OpenShift networking layer, this reference architecture uses the default Open vSwitch 'flat' pod network.

DEPLOYMENT

While it is not in the scope of this document to explain how to deploy the Red Hat OpenShift Container Platform, there are several deployment elements that need to be explicitly detailed as to the creation of the specific infrastructure that was used for this reference architecture.

Pure Storage Red Hat Best Practices

To ensure that the FlashArray connections are optimal, it is necessary to install the latest **device-mapper-multipath** package as well as the latest **iscsi-initiator-utils** package, and then enable both the **multipath** and **iscsid** daemons, to ensure they persist after any reboots. More details can be found in the Pure Storage Knowledge Base article on Linux Recommendations.

> 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.

Configuring Docker Storage

As part of the host preparation stage of an OpenShift deployment, there is a requirement for Docker storage which will hold, temporarily, containers and images. This is separate from any persistent storage required by applications running within OpenShift.

The default storage backend for a bare-metal Red Hat Enterprise Linux host is a thin pool on a loopback device, however, this is not appropriate, or supported, for a production deployment. Therefore, a thin pool logical volume must be created for each host in the deployment (except the bastion) and Docker must be reconfigured to use this logical volume on each host.

> The following steps should be taken to configure a Pure Storage volume to act as the local Docker storage. These steps must be implemented on all nodes in the OpenShift cluster.

1. Create a host object for the node in the OpenShift cluster using the iSCSI IQN for the host port.
2. Create a 20GiB volume on your FlashArray and connect to its associated host.
3. Rescan the host to ensure that the new iSCSI volume is available to the operating system.
4. Check that the host is connected to the FlashArray iSCSI ports using the **iscsiadm** command, ensuring these logins persist over reboot.
5. Check the multipath device name for the new iSCSI volume using the **multipath** command. For this example we will assume the multipath device created is **/dev/dm-3**
6. Modify the file **/etc/sysconfig/docker-storage-setup** to contain the following:

```
# cat <<EOF > /etc/sysconfig/docker-storage-setup
DEVS=/dev/dm-3
WRITE_SIGNATURES=true
```

```
VG=docker-vol
EOF
```

7. Run the **docker-storage-setup** command.
8. Reinitialize the docker process.

```
# systemctl stop docker
# rm -rf /var/lib/docker/*
# systemctl start docker
```

9. Ensure that the thin pool has been created correctly using both the **docker info** and **lvs** commands.

Configure Docker Registry Storage

As part of the deployment phase of an OpenShift Container Platform, you must enable a local Docker registry and define its storage location and configuration.

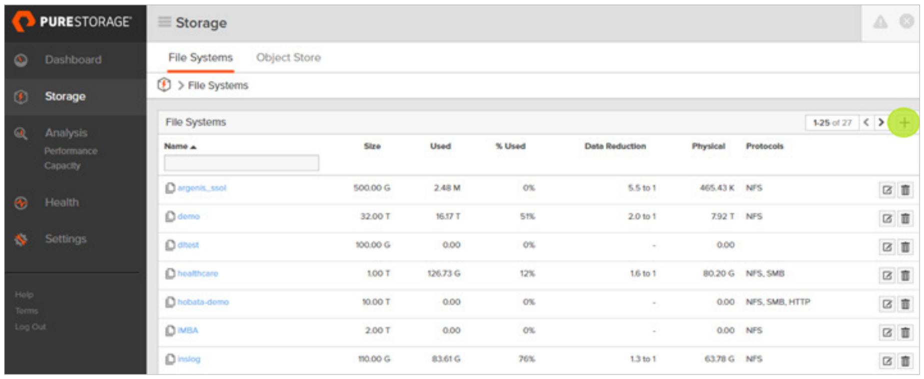
This registry storage can be located on a local NFS host group directory, an external NFS host, an OpenStack platform (using the Glance project), or an S3 storage solution (public or private).

Here we give details on how to configure a Pure Storage FlashBlade to provide the backing store for the Docker Registry using NFS.

NFS DIRECTORY SHARE

The following steps describe how to create an NFS share on FlashBlade and how to use this information in an OpenShift deployment process.

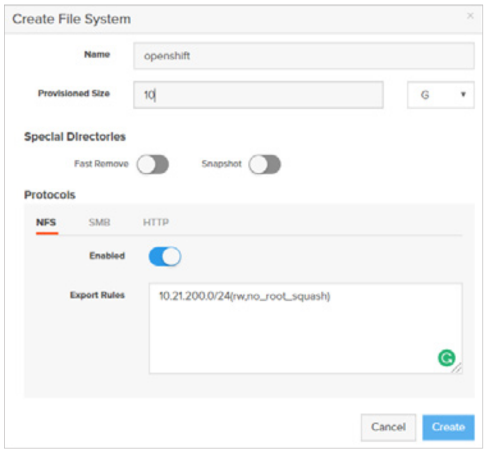
- Create an NFS share from the FlashBlade GUI.
  – Navigate to the Storage > File Systems page:



Name ▲	Size	Used	% Used	Data Reduction	Physical	Protocols
argenis_ssol	500.00 G	2.48 M	0%	5.5 to 1	465.43 K	NFS
demo	32.00 T	16.17 T	51%	2.0 to 1	7.92 T	NFS
dttest	100.00 G	0.00	0%	-	0.00	
healthcare	1.00 T	126.73 G	12%	1.6 to 1	80.20 G	NFS, SMB
hobata-demo	10.00 T	0.00	0%	-	0.00	NFS, SMB, HTTP
IMBA	2.00 T	0.00	0%	-	0.00	NFS
inslog	110.00 G	83.61 G	76%	1.3 to 1	63.78 G	NFS

FIGURE 2. Storage > File Systems screen in the FlashArray GUI

  – Select the + icon in the top right of the pane to get the following popup:


Create File System

Name: openshift

Provisioned Size: 10 G

Special Directories

Fast Remove | Snapshot

Protocols

NFS | SMB | HTTP

Enabled

Export Rules: 10.21.200.0/24(rw,no_root_squash)

Cancel | Create

FIGURE 3. Create File System popup

   – Give a **Name** for the NFS share and a capacity. Ensure that the **Enabled** switch is selected on and, if necessary, add an export rule to ensure that all OpenShift nodes can access this share. In this example, we are creating a share called **openshift** with a size of 10GiB.

- Temporarily mount your newly created NFS on any server that has access to it, and create a subdirectory which will be referenced as **volume name** in the next bullet point. In this case, we will create a directory called **pure-ra**.

```
# mount -t nfs <NFS service IP>:/openshift /mnt
# mkdir /mnt/pure-ra
# chmod a+w /mnt/pure-ra
# umount /mnt
```

- Use the information from FlashBlade to populate the OpenShift deployment Ansible inventory file with the following details (here using values from the above examples):

```
openshift_hosted_registry_storage_kind=nfs
openshift_hosted_registry_storage_access_modes=['ReadWriteMany']
openshift_hosted_registry_storage_host=<IP address of NFS data service>
openshift_hosted_registry_storage_nfs_directory=/openshift
openshift_hosted_registry_storage_volume_name=pure-ra
openshift_hosted_registry_storage_volume_size=10Gi
```

- After a successful deployment of OpenShift, you will see both a Persistent Volume and a Persistent Volume Claim have been automatically created using your NFS share:

```
# oc get pv
NAME            CAPACITY  ACCESSMODES  RECLAIMPOLICY  STATUS  CLAIM                 STORAGECLASS  REASON  AGE
pure-ra-volume  100Gi     RWX          Retain         Bound   default/pure-ra-claim                        6h
```

```
# oc get pvc
NAME           STATUS    VOLUME           CAPACITY   ACCESSMODES   STORAGECLASS   AGE
pure-ra-claim  Bound     pure-ra-volume   100Gi      RWX                          6h
```

PERSISTENT STORAGE

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

The Pure Service Orchestrator plugin provides both file- and block-based **Storage Classes**, provisioned from either FlashArray or FlashBlade storage devices.

> To make these StorageClasses available to your OpenShift cluster, you must install the Pure Service Orchestrator OpenShift plugin.

Pure Service Orchestrator Installation

Installation and configuration of PSO is simple and requires only a few steps, which are described in the Docker Store location of the Pure Service Orchestrator.

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 Pure Storage Linux Best Practices.

PLUGIN INSTALLATION

Pure Service Orchestrator manages the installation of all required files across your OpenShift environment by using a DaemonSet to perform cross-node installation. The DaemonSet runs a pod on each appropriate 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. Perform the following steps to install Pure Service Orchestrator:

1. Install Helm[1] in your OpenShift deployment
2. Add the pure repo to Helm:

```
# oc adm policy add-role-to-user cluster-admin "system:serviceaccount:${TILLER_NAMESPACE}:tiller"
# helm repo add pure http://purestorage.github.io/helm-charts
# helm repo update
# helm search pure-k8s-plugin
```

[1] Details of a Red Hat approved method to install Helm can be found at https://blog.openshift.com/getting-started-helm-openshift/

3. Update the PSO configuration file

To enable Pure Service Orchestrator to communicate with your Pure Storage backend arrays, it is required to update a 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 must be supplied for each FlashBlade.

Take a copy of the values.yaml provided by the Helm Chart[2] 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"
      Labels:
        rack: "22"
        env: "prod"
    - 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"
      Labels:
        rack: "7b"
        env: "dev"
    - MgmtEndPoint: "1.2.3.8"
      APIToken: "T-d4925090-c9bf-4033-8537-d24ee5669135"
      NfsEndPoint: "1.2.3.9"
      Labels:
        rack: "6a"
```

Ensure that the values you enter are correct for your own Pure Storage devices. Additionally, in the orchestrator section of the yaml file, change the orchestrator name for openshift and amend the **flexPath** parameter as follows:

```
orchestrator:
  name: openshift
```

[2] Or download from https://raw.githubusercontent.com/purestorage/helm-charts/master/pure-k8s-plugin/values.yaml

3. Configure labels

You will see in the above example for the arrays that entries have one or more labels assigned to them. Labels can be used to filter the list of backends. Labels are arbitrary (key, value) pairs that can be added to any backend, as seen in the example above. More than one backend can have the same (key, value) pair. When creating a new volume, label (key = value) pairs can be specified to filter the list of backends to a given set. The plugin also provides the following well known labels that can be used. There is no requirement to add this into your **values.yaml** file.

- **purestorage.com/backend:** Holds the value file for FlashBlades and block for FlashArrays.
- **purestorage.com/hostname:** Holds the host name of the backend.
- **purestorage.com/id:** Holds the ID of the backend.
- **purestorage.com/family:** Holds either FlashArray or FlashBlade

4. Create Security Context

As this plugin needs to mount external volumes to containers, there is a requirement under OpenShift security rules for the provisioner pod created by the plugin to use the hostPath volume plugin. To do this without having to grant everyone access to privileged SCCs, such as privileged or hostaccess, it is recommended to create a new SCC and grant this to all users.[3]

Define the new SCC in a YAML file:

```
kind: SecurityContextConstraints
apiVersion: v1
metadata:
  name: hostpath
allowPrivilegedContainer: true
allowHostDirVolumePlugin: true
runAsUser:
  type: RunAsAny
seLinuxContext:
  type: RunAsAny
fsGroup:
  type: RunAsAny
supplementalGroups:
  type: RunAsAny
```

Now use **oc create** and pass the YAML file to create the new SCC:

```
# oc create -f hostpath-scc.yaml
```

[3] https://docs.openshift.com/container-platform/3.11/admin_guide/manage_scc.html#use-the-hostpath-volume-plugin

Finally, grant access to this SCC to all users

```
# oc adm policy add-scc-to-group hostpath system:authenticated
```

This is only one option. There are many other ways to ensure the PSO plugin works under different SCC configurations, and you must ensure that the method you choose is appropriate to your specific environment.

5. Install the plugin

To ensure proper security is maintained within your OpenShift cluster, it is recommended that the plugin be installed in its own project. Create this project using the following command:

```
# oc new-project pso
```

It is advisable to perform a 'dry run' installation to ensure that your YAML file is correctly formatted:

```
# helm install --namespace pso pure-storage-driver pure/pure-k8s-plugin -f <your_own_dir>/<your_own_
values>.yaml --dry-run --debug
```

Perform the actual install. Unless otherwise specified, the install will occur in the **default** namespace.

```
# helm install --namespace pso pure-storage-driver pure/pure-k8s-plugin -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 --namespace pso pure-storage-driver pure/pure-k8s-plugin -f <your_own_dir>/<your own
values>.yaml --set flasharray.sanType=FC,namespace.pure=k8s_xxx
```

It is recommended 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 INSTALLATION

After running the installer, we can check to ensure that all the necessary components have been correctly installed using the following commands:

```
# oc get deployments
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
pure-provisioner   1          1          1             1            2m
# oc get sc
NAME             TYPE
pure             pure-provisioner
```

```
pure-block        pure-provisioner
pure-file         pure-provisioner
# oc get pods
NAME                                   READY     STATUS    RESTARTS   AGE
pure-flex-9sphg                       1/1       Running   0          1h
pure-flex-b7bt9                       1/1       Running   0          1h
pure-flex-dhc4d                       1/1       Running   0          1h
pure-flex-f5xtk                       1/1       Running   0          1h
pure-flex-kcp4n                       1/1       Running   0          1h
pure-flex-rg8q6                       1/1       Running   0          1h
pure-flex-rqsxq                       1/1       Running   0          1h
pure-flex-xmn72                       1/1       Running   0          1h
pure-provisioner-2785090122-78kmf    1/1       Running   0          1h
# oc get daemonset
NAME        DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE-SELECTOR                             AGE
pure-flex   8         8         8       8            8           node-role.kubernetes.io/compute=true      1h
```

We can see the dynamic provisioner running as a single pod and multiple pure-flex pods, one for each compute node in the cluster. These multiple pods are making sure the Pure Storage FlexVolume driver is healthy on each node, these being coordinated by the pure-flex daemonset.

SIMPLE POD DEPLOYMENT WITH A PURE STORAGE FLASHARRAY PERSISTENT VOLUME

To validate that Pure Service Orchestrator plugin has been configured and installed correctly, we can create a simple pod with the **Nginx** application 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. It is recommended that each application is deployed in its own project, therefore for this simple test we will first create a new project.

```
# oc new-project nginx
```

Persistent Volume Claim

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

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

Notice that the **storageClassName** is **pure-block**, which ensures we get a PV from a FlashArray backend. If we used **pure** as the value here this would also provide a FlashArray based PV.

Execute the following command:

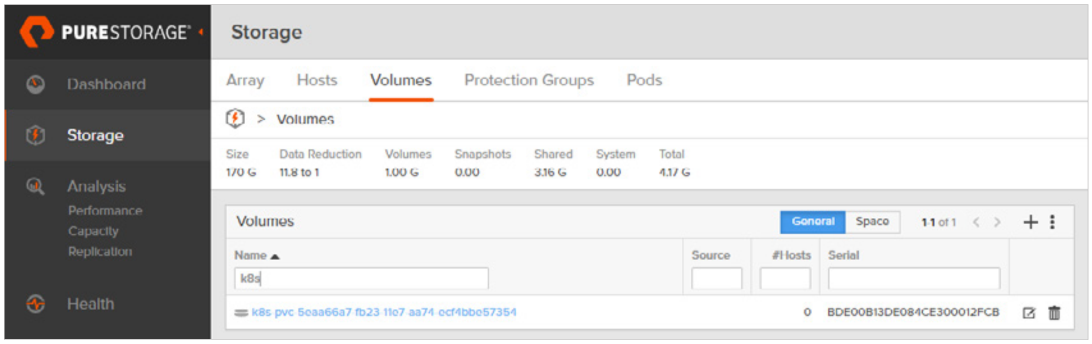
```
  # oc create -f nginx-pvc.yaml
```

This will create a PVC called **pure-claim** 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:

```
  # oc get pvc
  NAME          STATUS   VOLUME                                     CAPACITY   ACCESSMODES   STORAGECLASS   AGE
  pure-claim    Bound    pvc-5eaa66a7-fb23-11e7-aa74-ecf4bbe57354   10Gi       RWO           pure           1h
```

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

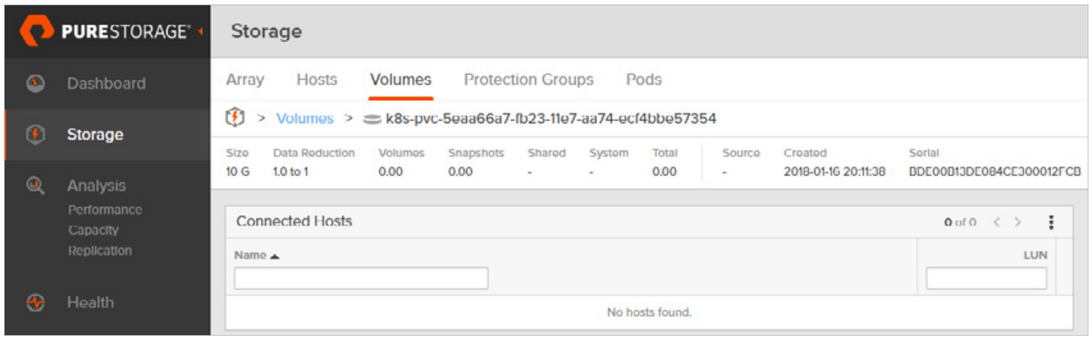


Size	Data Reduction	Volumes	Snapshots	Shared	System	Total
170 G	11.8 to 1	1.00 G	0.00	3.16 G	0.00	4.17 G

Name ▲	Source	# Hosts	Serial
k8s			
k8s-pvc-5eaa66a7-fb23-11e7-aa74-ecf4bbe57354		0	BDE00B13DE084CE300012FCB

FIGURE 4. FlashArray GUI Storage > Volumes pane

We can see that the volume name matches the PV name with a prefix of **k8s-**. This prefix is the **pure:namespace** parameter in the Helm **values.yaml** file, which defaults to **k8s**. 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.



> Volumes > k8s-pvc-5eaa66a7-fb23-11e7-aa74-ecf4bbe57354

Size	Data Reduction	Volumes	Snapshots	Shared	System	Total	Source	Created	Serial
10 G	1.0 to 1	0.00	0.00	-	-	0.00	-	2018-01-16 20:11:38	BDE00B13DE084CE300012FCB

Connected Hosts 0 of 0

Name ▲	LUN

No hosts found.

FIGURE 5. FlashArray GUI Storage > Volumes pane

Application Pod

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

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  volumes:
  - name: pure-vol
     persistentVolumeClaim:
         claimName: pure-claim
  containers:
  - name: nginx
     image: nginx
  volumeMounts:
  - name: pure-vol
     mountPath: /data
  ports:  pure
  - containerPort: 80
```

Execute the following command:

```
# oc create -f nginx-pod.yaml
```

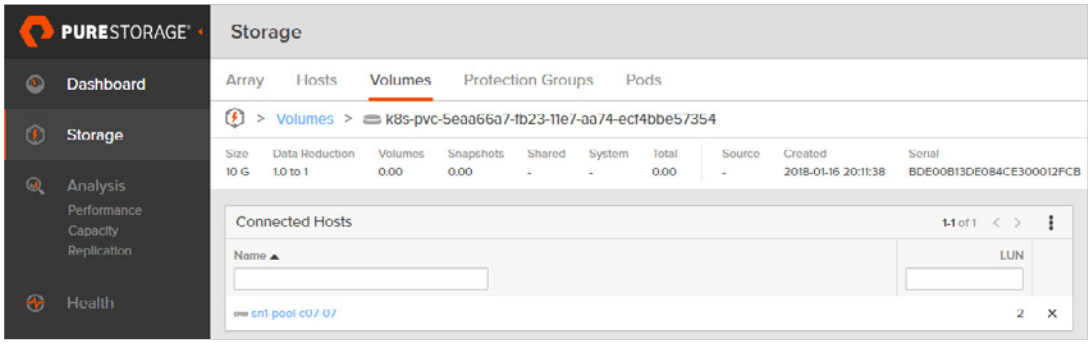
This will create a pod called **nginx** that will run the Nginx image, and the FlexVolume driver will mount the PV created earlier to the directory **/data** within the pod.

A lot of information can be gathered regarding the newly created pod – some useful information is highlighted below:

```
# oc describe pod nginx
Name:           nginx
Namespace:      nginx
Node:           sn1-pool-c07-07.puretec.purestorage.com/10.21.200.117
Start Time:     Mon, 18 Mar 2019 08:50:28 -0800
Labels:         <none>
Annotations:    openshift.io/scc=privileged
Status:         Running
IP:             10.131.0.41
Containers:
  nginx:
    Container ID:       docker://9623179a0427b4253cc9a71f629cc236d172f06936809117af26758f1e6b4073
    Image:              nginx
    Image ID:           docker-pullable://docker.io/nginx@
sha256:2ffc60a51c9d658594b63ef5acfac9d92f4e1550f633a3a16d898925c4e7f5a7
    Port:               80/TCP
    State:              Running
      Started:          Mon, 18 Mar 2019 08:50:35 -0800
    Ready:              True
    Restart Count:      0
    Environment:        <none>
    Mounts:
      /data from pure-vol (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-dcbmd (ro)
Conditions:
  Type           Status
  Initialized    True
  Ready          True
  ContainerReady True
  PodScheduled   True
Volumes:
  pure-vol:
    Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace)
    ClaimName:  pure-claim
    ReadOnly:   false
  default-token-dcbmd:
    Type:       Secret (a volume populated by a Secret)
```

```
      SecretName:  default-token-dcbmd
      Optional:    false
QoS Class:         BestEffort
Node-Selectors:  node-role.kubernetes.io/compute=true
Tolerations:     <none>
Events:          <none>
```

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



> Volumes > k8s-pvc-5eaa66a7-fb23-11e7-aa74-ecf4bbe57354

Size	Data Reduction	Volumes	Snapshots	Shared	System	Total	Source	Created	Serial
10 G	1.0 to 1	0.00	0.00	-	-	0.00	-	2018-01-16 20:11:38	BDE00B13DE084CE300012FCB

Connected Hosts 1-1 of 1

Name ▲	LUN
sn1 pool c07 07	2

FIGURE 6. FlashArray GUI Storage > Volumes pane

We can confirm the Nginx application is working by performing a simple web call to the pod IP address:

```
# curl http://10.131.0.41
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
```

```
<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

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 installed and configured correctly and 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. Here are are still using the previously created project nginx, but you can confirm you are still within this project by using the following command:

```
# oc project nginx
```

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
```

Note that we are specifically calling the **storageClassName** to be **pure-file** to ensure we get a PV created on a FlashBlade backend.

Execute the following command:

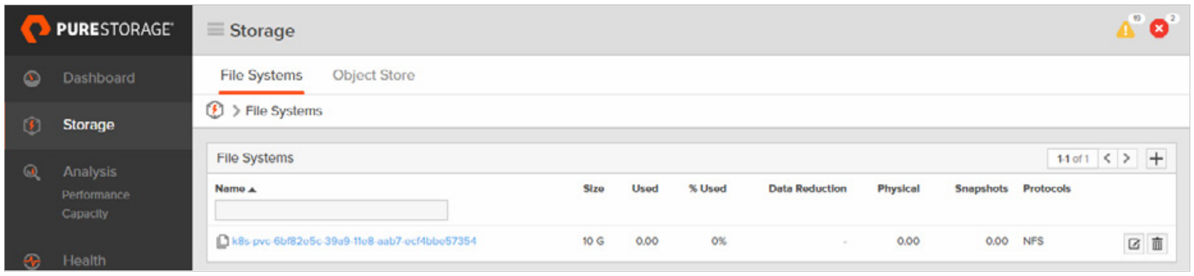
```
# oc create -f nginx-nfs-pvc.yaml
```

This will create a PVC called **pure-nfs-claim** 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:

```
# oc get pvc
NAME            STATUS  VOLUME                                    CAPACITY  ACCESSMODES  STORAGECLASS  AGE
pure-nfs-claim  Bound   pvc-6bf82e5c-39a9-11e8-aab7-ecf4bbe57354  10Gi      RWX          pure-file     3m
```

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



Name ▲	Size	Used	% Used	Data Reduction	Physical	Snapshots	Protocols
k8s-pvc-6bf82e5c-39a9-11e8-aab7-ecf4bbe57354	10 G	0.00	0%	-	0.00	0.00	NFS

FIGURE 7. FlashArray GUI Storage > File Systems pane

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

Ensure NFS Write Access

By default, SELinux does not allow writing from pods to external NFS shares, therefore the following commands need to run on each node:

```
# setsebool -P virt_sandbox_use_nfs on
# setsebool -P virt_use_nfs on
```

A simple way to do this is to run the following Ansible command from your Ansible Bastion host:

```
# ansible -i <openshift deployment inventory file> nodes -m raw -a 'setsebool -P virt_sandbox_use_nfs on;
setsebool -P virt_use_nfs on'
```

Application Pod

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

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

Execute the following command:

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

This will create a pod called **nginx-pod-nfs** that will run the Nginx image and the **FlexVolume** 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; some useful information is highlighted below:

```
# oc describe pod nginx-nfs
Name:           nginx-nfs
Namespace:      nginx
Node:           sn1-pool-c07-10.puretec.purestorage.com/10.21.200.120
Start Time:     Mon, 18 Mar 2019 12:18:55 -0800
Labels:         <none>
Annotations:    openshift.io/scc=privileged
Status:         Running
IP:             10.131.2.66
```

```
Containers:
  nginx-nfs:
    Container ID:       docker://e4467f968215163f0501c56b210e2a79a3a9a9ce1c5a641333ea70ecd783856f
    Image:              nginx
    Image ID:           docker-pullable://docker.io/nginx@
sha256:d0468eaec1ef818af05f85ac00e484fd5a2ae75dd567dc9f7ccf5f68a60351fb
    Port:               80/TCP
    State:              Running
      Started:          Mon, 18 Mar 2019 12:19:01 -0800
    Ready:              True
    Restart Count:      0
    Environment:        <none>
    Mounts:
      /usr/share/nginx/html from pure-nfs (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-nskqk (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainerReady    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-nskqk:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-nskqk
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  node-role.kubernetes.io/compute=true
Tolerations:     <none>
```

Additonal Application Pod

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

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: busybox-nfs
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:

```
# oc 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; some useful information is highlighted below:

```
# oc describe pod busybox-nfs
Name:           busybox-nfs
Namespace:      nginx
Node:           sn1-pool-c07-08.puretec.purestorage.com/10.21.200.118
Start Time:     Mon, 18 Mar 2019 12:23:45 -0800
Labels:         <none>
Annotations:    openshift.io/scc=privileged
Status:         Running
IP:             10.129.2.101
Containers:
  busybox-nfs:
    Container ID:       docker://87b0a962f7d88c11510ab7ec6fbcb762a535a39ceb1c8453e06874423c3f436c
    Image:              busybox
    Image ID:           docker-pullable://docker.io/busybox@
sha256:58ac43b2cc92c687a32c8be6278e50a063579655fe3090125dcb2af0ff9e1a64
    Port:               <none>
    Command:
      sleep
```

```
        60000
      State:              Running
        Started:          Mon, 18 Mar 2019 12:23:54 -0800
      Ready:              True
      Restart Count:      0
      Environment:        <none>
      Mounts:
        /usr/share/busybox from pure-nfs-2 (rw)
        /var/run/secrets/kubernetes.io/serviceaccount from default-token-nskqk (ro)
  Conditions:
    Type              Status
    Initialized       True
    Ready             True
    ContainerReady    True
    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-nskqk:
      Type:       Secret (a volume populated by a Secret)
      SecretName: default-token-nskqk
      Optional:   false
  QoS Class:       BestEffort
  Node-Selectors:  node-role.kubernetes.io/compute=true
  Tolerations:     <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.

UPDATING PURE SERVICE ORCHESTRATOR CONFIGURATION

As your OpenShift platform scales with increased demand from applications, workflows, and users, there will inevitably be 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 Pure Service Orchestrator, adding additional storage backends or changing labels is seamless.

The process is as simple as updating your values YAML file with new labels or adding new FlashArray or FlashBlade access information and then running this single command:

```
# helm upgrade pure-storage-driver pure/pure-k8s-plugin -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.

ADDING CLUSTER NODES TO OPENSHIFT

This reference architecture used three master nodes, two infrastructure nodes, and three application nodes. This cluster is large enough to provide sufficient resources to run a few simple applications (see Appendix 2). However, as a cluster becomes more utilized, it may be necessary to provide additional application or infrastructure nodes to support additional resource requirements.

The current default scalability limits for OpenShift 3.11 are 250 pods per node. The maximum number of nodes in a single OpenShift 3.11 cluster is 2,000. However, if you are reaching a cluster of this size, you should be using multiple backend storage arrays to provide your persistent volumes.

There are processes for adding additional nodes to an OpenShift cluster, and they are well documented within the main Red Hat OpenShift documentation set, but it is important to cover how to ensure that additional nodes have the ability to utilize the Pure Storage arrays as providers of stateful storage.

If required, you can configure the Docker local registry for each new node, as described earlier in this document, before performing the OpenShift node add procedures.

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 OpenShift cluster will cause the DaemonSet to create a new pure-flex pod on the new node and install the plugin correctly.

CONCLUSION

The Red Hat OpenShift Container Platform provides a Kubernetes-based, production-ready infrastructure foundation to simplify deployment processes, and provides a stable, highly-available platform on which to run production applications.

With the growth of applications and deployments that require a platform that can also provide an underlying stateful storage solution, the Pure Service Orchestrator 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 1: ANSIBLE DEPLOYMENT INVENTORY

The following inventory was used in the deployment of the OpenShift cluster used in this Reference Architecture. There are a number of key items which are necessary for the correct operation of the configured OpenShift cluster that have been described in earlier sections of this document.

```
[OSEv3:children]
masters
etcd
nodes
lb

[OSEv3:vars]
openshift_deployment_type=openshift-enterprise
openshift_release=v3.11

openshift_cluster_network_cidr=10.128.0.0/14
openshift_portal_net=172.30.0.0/16
openshift_master_api_port=8443
openshift_master_console_port=8443

openshift_master_dynamic_provisioning_enabled=True
openshift_master_cluster_method=native
openshift_master_cluster_hostname=sn1-pool-c07-11.puretec.purestorage.com
openshift_master_cluster_public_hostname=sn1-pool-c07-11.puretec.purestorage.com

openshift_master_identity_providers=[{'name': 'htpasswd_auth', 'login': 'true', 'challenge': 'true', 'kind':
'HTPasswdPasswordIdentityProvider'}]

openshift_master_htpasswd_users={'admin': '$apr1$3dmqnWzp$Ao46LLSg.otFvtZoTEJ7l0', 'developer': '$apr1$GiIIk.
Ke$5DQqHlrwcsIsWGcMaLtgz1'}

openshift_master_default_subdomain=apps.puretec.purestorage.com
oreg_url=registry.redhat.io/openshift3/ose-${component}:${version}
oreg_auth_user=<INSERT RED HAT USERNAME>
oreg_auth_password=<INSERT RED HAT PASSWORD>

# Configure internal registry to use Pure Storage FlashBlade NFS share
openshift_hosted_registry_storage_kind=nfs
```

```
openshift_hosted_registry_storage_access_modes=['ReadWriteMany']
openshift_hosted_registry_storage_host=10.21.97.48
openshift_hosted_registry_storage_nfs_directory=/openshift
openshift_hosted_registry_storage_volume_name=pure-ra
openshift_hosted_registry_storage_volume_size=100Gi

ansible_ssh_user=root

[masters]
sn1-pool-c07-0[3:5].puretec.purestorage.com

[etcd]
sn1-pool-c07-0[3:5].puretec.purestorage.com

[nodes]
sn1-pool-c07-0[3:5].puretec.purestorage.com openshift_node_group_name='node-config-master'
sn1-pool-c07-0[6:7].puretec.purestorage.com openshift_node_group_name='node-config-infra'
sn1-pool-c07-0[8:9].puretec.purestorage.com openshift_node_group_name='node-config-compute'
sn1-pool-c07-10.puretec.purestorage.com openshift_node_group_name='node-config-compute'

[lb]
sn1-pool-c07-11.puretec.purestorage.com
```

APPENDIX 2: OPENSHIFT CATALOG TO DEPLOY SIMPLE APPLICATIONS

Whilst the above section details some more complex implementations of applications using persistent storage, the OpenShift GUI provides a simple catalog of applications with click-through deployment.

Here is an example of using this catalog to deploy a single MongoDB pod with persistent storage provided by a Pure Storage FlashArray, assuming the **StorageClass** pure-block has been set as the default. Setting a storageClass to be a default is performed as follows:

```
# oc patch storageclass pure-block -p '{"metadata": {"annotations": {"storageclass.kubernetes.io/is-default-
class": "true"}}}'
```

First, we log into the OpenShift Container Platform GUI and navigate to the **Catalog**, then select the **Databases** sub-page and then the **Mongo** tab, and finally the **MongoDB (Persistent)** icon, as shown in the following screenshots.

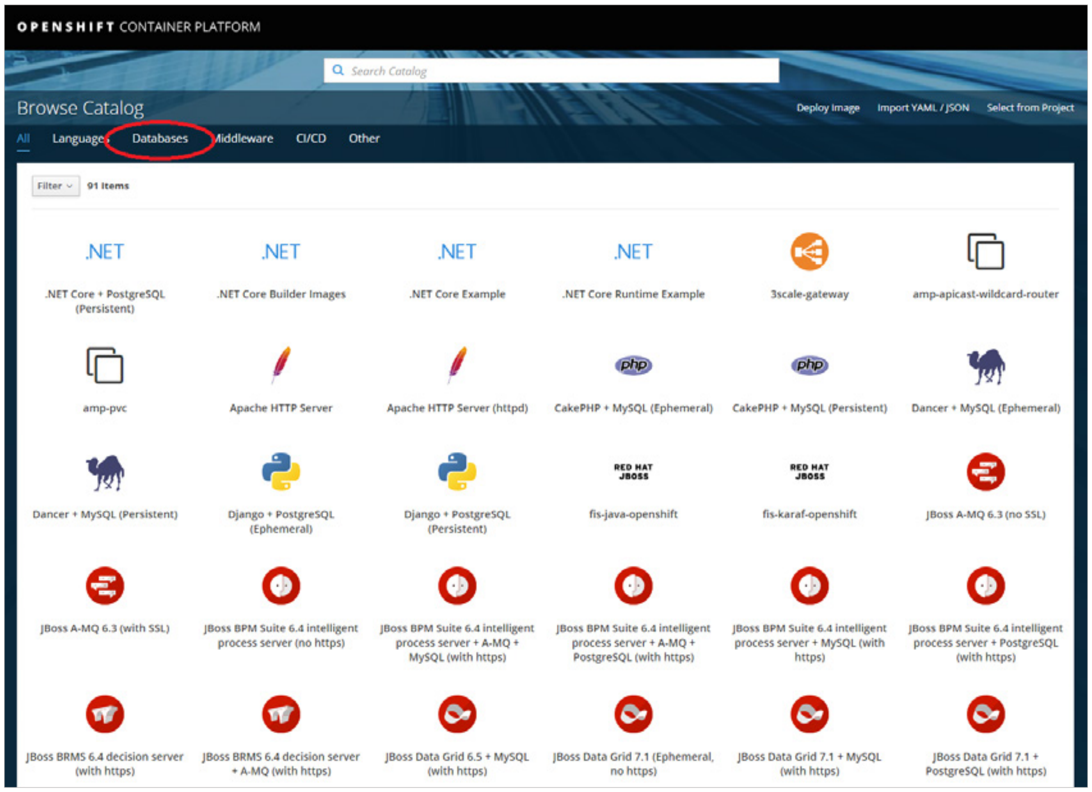

OPENSHIFT CONTAINER PLATFORM
Search Catalog
Browse Catalog
Deploy Image
Import YAML / JSON
Select from Project
All
Languages
Databases
Middleware
CI/CD
Other
Filter
91 Items
.NET Core + PostgreSQL (Persistent)
.NET Core Builder Images
.NET Core Example
.NET Core Runtime Example
3scale-gateway
amp-apicast-wildcard-router
amp-pvc
Apache HTTP Server
Apache HTTP Server (httpd)
CakePHP + MySQL (Ephemeral)
CakePHP + MySQL (Persistent)
Dancer + MySQL (Ephemeral)
Dancer + MySQL (Persistent)
Django + PostgreSQL (Ephemeral)
Django + PostgreSQL (Persistent)
fis-java-openshift
fis-karaf-openshift
JBoss A-MQ 6.3 (no SSL)
JBoss A-MQ 6.3 (with SSL)
JBoss BPM Suite 6.4 intelligent process server (no https)
JBoss BPM Suite 6.4 intelligent process server + A-MQ + MySQL (with https)
JBoss BPM Suite 6.4 intelligent process server + A-MQ + PostgreSQL (with https)
JBoss BPM Suite 6.4 intelligent process server + MySQL (with https)
JBoss BPM Suite 6.4 intelligent process server + PostgreSQL (with https)
JBoss BRMS 6.4 decision server (with https)
JBoss BRMS 6.4 decision server + A-MQ (with https)
JBoss Data Grid 6.5 + MySQL (with https)
JBoss Data Grid 7.1 (Ephemeral, no https)
JBoss Data Grid 7.1 + MySQL (with https)
JBoss Data Grid 7.1 + PostgreSQL (with https)

FIGURE 12. Main Openshift Application Catalog

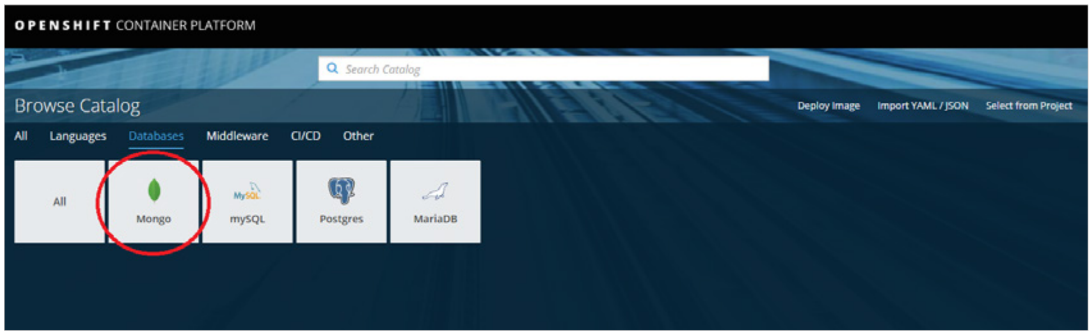


FIGURE 13. Database Sub-Catalog

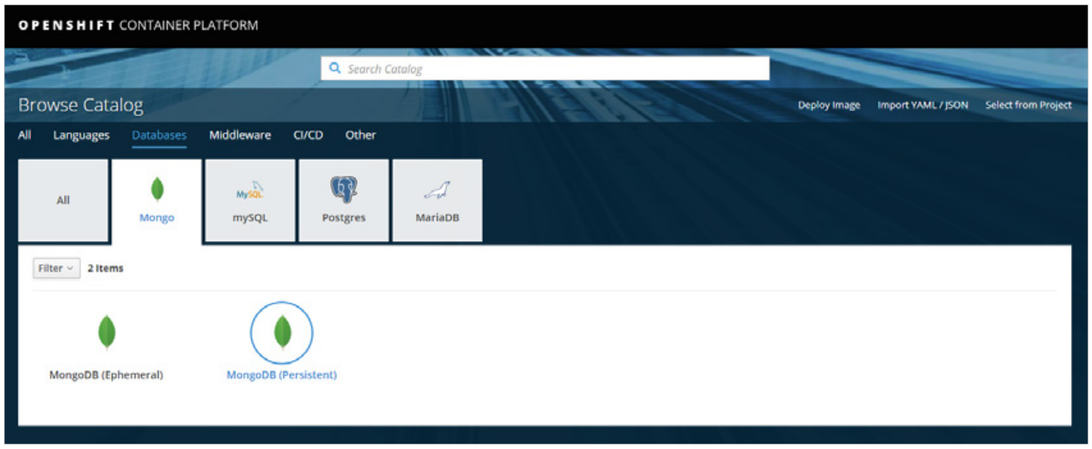
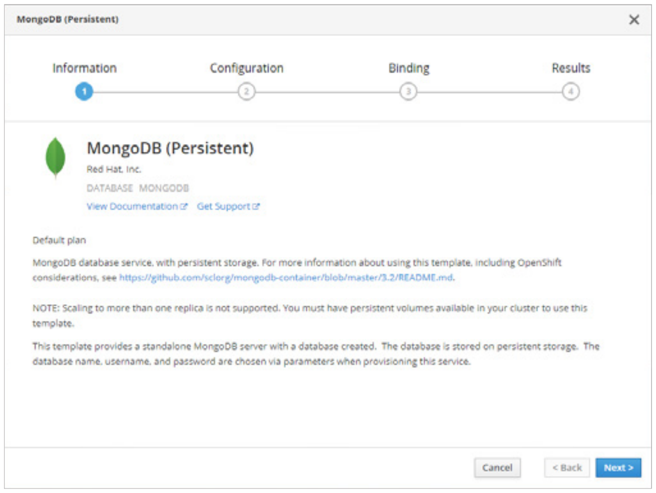


FIGURE 14. MongoDB Options

The final selection will create pop-up windows to describe the deployment plan for MongoDB with Persistent Storage. Go through these steps as follows:



MongoDB (Persistent)

Red Hat, Inc.

DATABASE MONGODB

View Documentation Get Support

Default plan

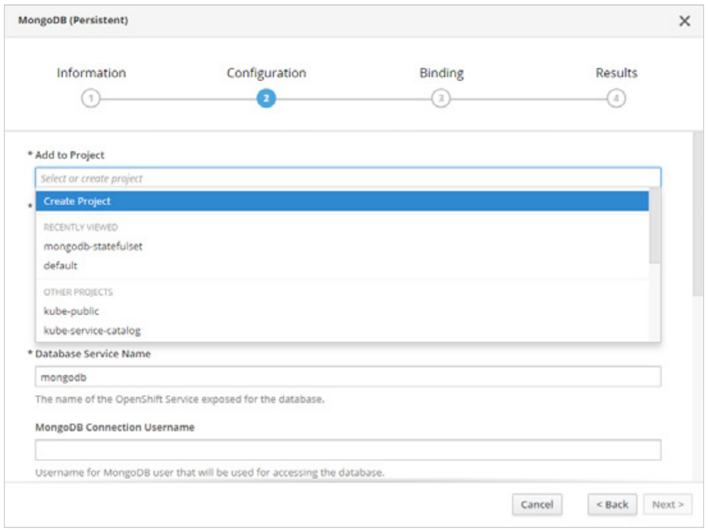
MongoDB database service, with persistent storage. For more information about using this template, including OpenShift considerations, see https://github.com/sclorg/mongodb-container/blob/master/3.2/README.md.

NOTE: Scaling to more than one replica is not supported. You must have persistent volumes available in your cluster to use this template.

This template provides a standalone MongoDB server with a database created. The database is stored on persistent storage. The database name, username, and password are chosen via parameters when provisioning this service.

FIGURE 15. MongoDB (Persistent) Information pane

Select **Next** to bring up the configuration step in the deployment process.


MongoDB (Persistent)

Information · Configuration · Binding · Results

* Add to Project

Select or create project

Create Project

RECENTLY VIEWED

mongodb-statefulset

default

OTHER PROJECTS

kube-public

kube-service-catalog

* Database Service Name

mongodb

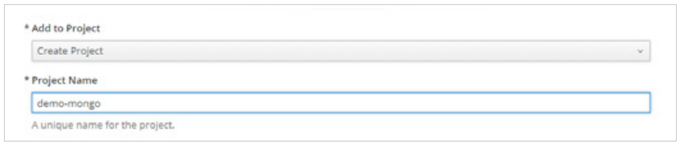
The name of the OpenShift Service exposed for the database.

MongoDB Connection Username

Username for MongoDB user that will be used for accessing the database.

FIGURE 16. MongoDB (Persistent) Configuration pane

In **Add to Project**, select **Create New Project** and then complete the **Project Name** field:


* Add to Project

Create Project

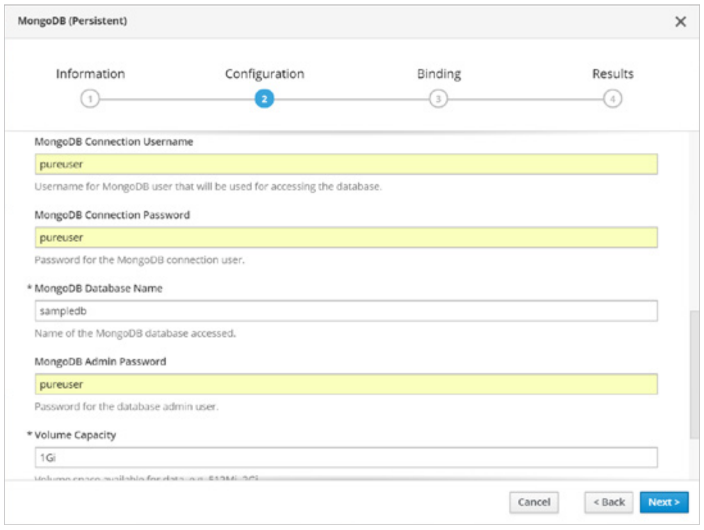
* Project Name

demo-mongo

A unique name for the project.

FIGURE 17. MongoDB Add to Project pane

Scroll down the configuration window and complete the required fields for **Connection Username** and **Password**, and **Admin Password**.


MongoDB (Persistent)

Information · Configuration · Binding · Results

MongoDB Connection Username

pureuser

Username for MongoDB user that will be used for accessing the database.

MongoDB Connection Password

pureuser

Password for the MongoDB connection user.

* MongoDB Database Name

sampledb

Name of the MongoDB database accessed.

MongoDB Admin Password

pureuser

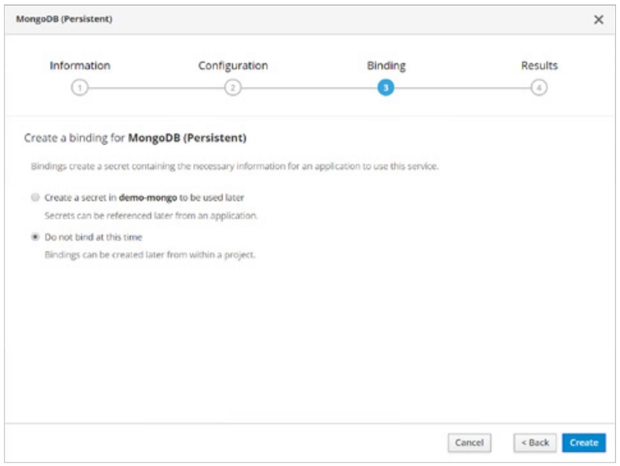
Password for the database admin user.

* Volume Capacity

1Gi

FIGURE 18. MongoDB (Persistent) Configuration pane

Leave the **Volume Capacity** at the default setting of **1 Gi** for this example and then select **Next** to move to the **Bindings** page:



Create a binding for **MongoDB (Persistent)**

Bindings create a secret containing the necessary information for an application to use this service.

Create a secret in **demo-mongo** to be used later [✓ Do not bind at this time]

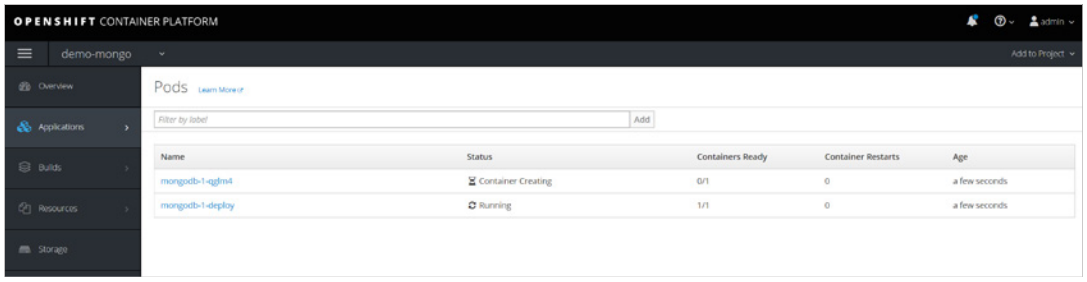
Secrets can be referenced later from an application.

Bindings can be created later from within a project.

FIGURE 19. MongoDB (Persistent) Bindings pane

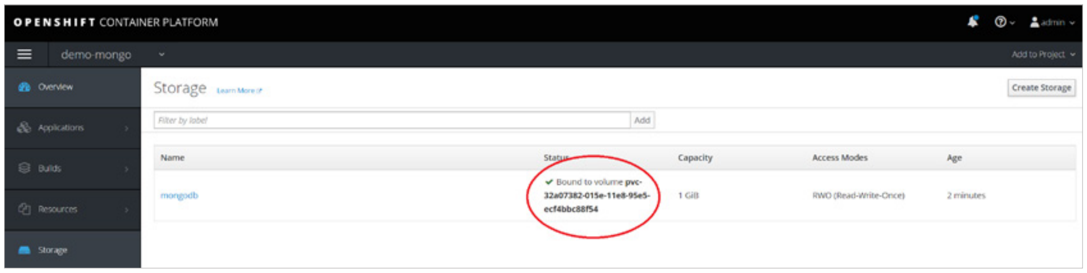
Leave this as the default and complete the application configuration by selecting **Create**.

We can now follow, through the OpenShift and Pure Storage FlashArray GUIs, the deployment of the MongoDB application using a deploy pod and an application pod, and the creation of persistent storage.



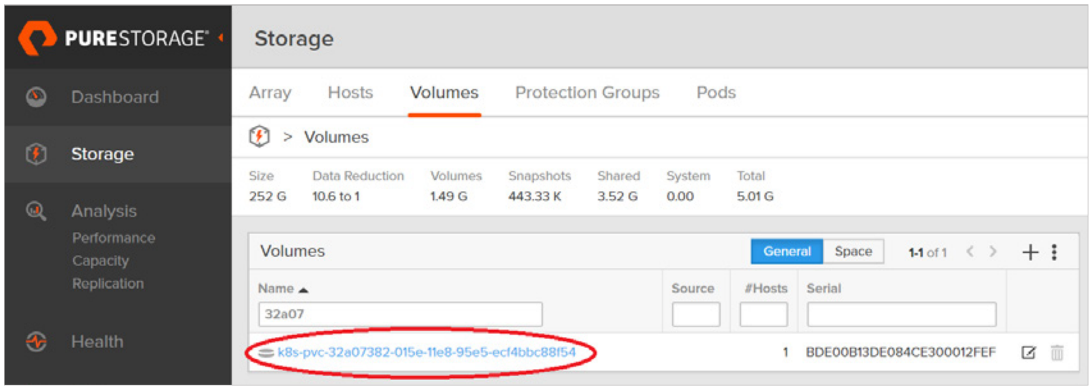
Name	Status	Containers Ready	Container Restarts	Age
mongodb-1-qglm4	Container Creating	0/1	0	a few seconds
mongodb-1-deploy	Running	1/1	0	a few seconds

FIGURE 20. Deploy and Application Pod builds in-progress



Name	Status	Capacity	Access Modes	Age
mongodb	✓ Bound to volume **pvc-32a07382-015e-11e8-95e5-ecf4bbc88f54**	1 GiB	RWO (Read-Write-Once)	2 minutes

FIGURE 21. Persistent Volume created



Storage

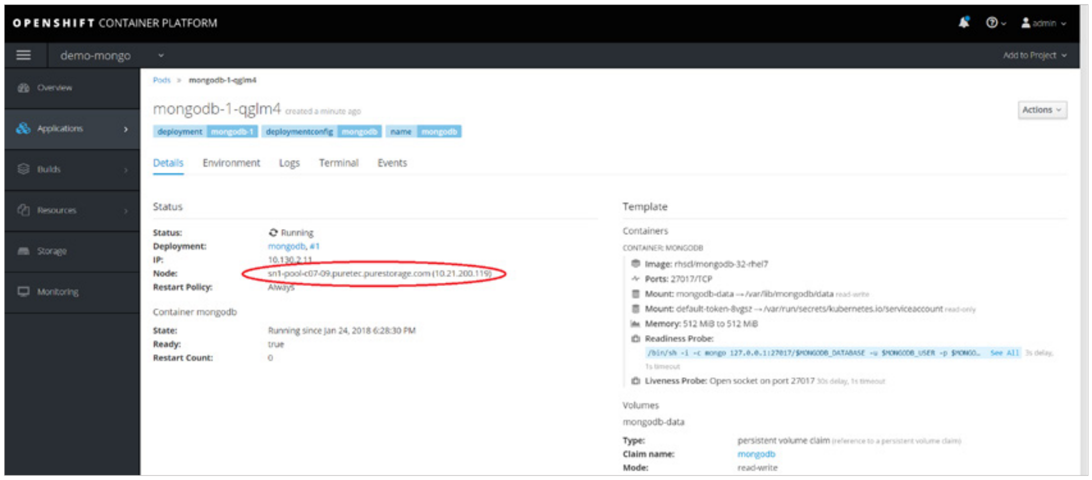
> Volumes

Size	Data Reduction	Volumes	Snapshots	Shared	System	Total
252 G	10.6 to 1	1.49 G	443.33 K	3.52 G	0.00	5.01 G

Volumes 1-1 of 1

Name ▲	Source	#Hosts	Serial
32a07			
k8s-pvc-32a07382-015e-11e8-95e5-ecf4bbc88f54		1	BDE00B13DE084CE300012FEF

FIGURE 22. Volume created on FlashArray



Pods > mongodb-1-qglm4

mongodb-1-qglm4 created a minute ago

deployment mongodb-1 deploymentconfig mongodb name mongodb

Status

Status: Running
Deployment: mongodb, #1
IP: 10.130.2.11
Node: sn1-pool-c07-09.puretec.purestorage.com (10.21.200.119)
Restart Policy: Always

Container mongodb

State: Running since Jan 24, 2018 6:28:30 PM
Ready: true
Restart Count: 0

Template

Containers

CONTAINER: MONGODB

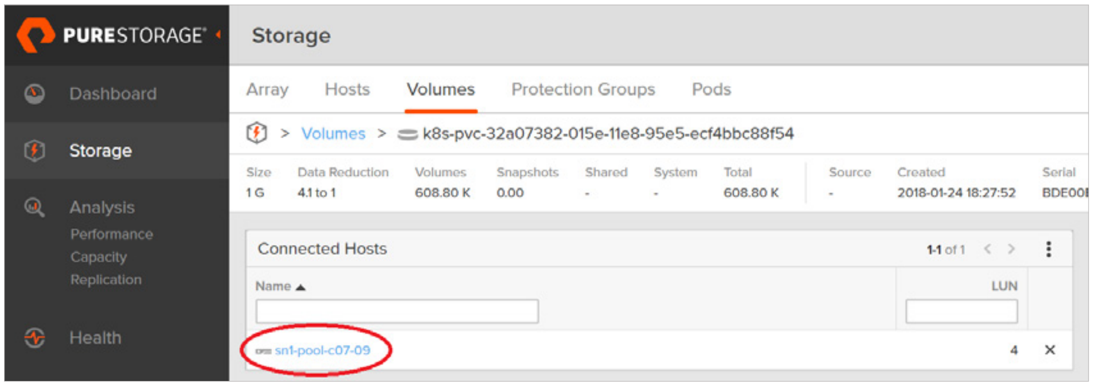
Image: rhscl/mongodb-32-rhel7
Ports: 27017/TCP
Mount: mongodb-data → /var/lib/mongodb/data read-write
Mount: default-token-8vgsz → /var/run/secrets/kubernetes.io/serviceaccount read-only
Memory: 512 MiB to 512 MiB
Readiness Probe:
`/bin/sh -i -c mongo 127.0.0.1:27017/$MONGODB_DATABASE -u $MONGODB_USER -p $MONGO… See All` 3s delay, 1s timeout
Liveness Probe: Open socket on port 27017 30s delay, 1s timeout

Volumes

mongodb-data

Type: persistent volume claim (reference to a persistent volume claim)
Claim name: mongodb
Mode: read-write

FIGURE 23. Application Pod detail showing container host



Storage

> Volumes > k8s-pvc-32a07382-015e-11e8-95e5-ecf4bbc88f54

Size	Data Reduction	Volumes	Snapshots	Shared	System	Total	Source	Created	Serial
1 G	4.1 to 1	608.80 K	0.00	-	-	608.80 K	-	2018-01-24 18:27:52	BDE00

Connected Hosts 1-1 of 1

Name ▲	LUN
sn1-pool-c07-09	4

FIGURE 24. FlashArray showing volume/host connection

APPENDIX 3: OPENSHIFT DEPLOYMENT LINKS

As this document does not cover details on deploying OpenShift, just specific details for implementing Pure Storage products in your deployment, here are some useful starting points to help your deployment of Red Hat OpenShift and some links to the Open Source product OpenShift Origin:

- https://docs.openshift.com/container-platform/3.11/install_config/index.html
- https://access.redhat.com/documentation/en-us/openshift_container_platform/3.11/html/installing_clusters/
- https://docs.openshift.org/latest/install_config/index.html
- https://docs.openshift.org/latest/minishift/index.html

ABOUT THE AUTHOR

As Director of New Stack Technologies, Simon Dodsley is helping direct and implement Open Source technologies in cloud, automation, and orchestration technologies within Pure Storage. Core items include best practices, reference architectures, and configuration guides.

With over 25 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 organizations, large and small, across Europe and the USA, as both customer and service provider. He also specializes in Data Migration methodologies, assisting customers in their Pure Storage transition.

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

ps_ra44p_platform-as-a-service_ltr_03