



OpenShift Dedicated 3

Architecture

OpenShift Dedicated 3 Architecture Information

OpenShift Dedicated 3 Architecture

OpenShift Dedicated 3 Architecture Information

Legal Notice

Copyright © 2018 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

Learn the architecture of OpenShift Dedicated 3 including the infrastructure and core components. These topics also cover authentication, networking and source code management.

Table of Contents

CHAPTER 1. OVERVIEW	6
1.1. WHAT ARE THE LAYERS?	6
1.2. WHAT IS THE OPENSIFT DEDICATED ARCHITECTURE?	7
1.3. HOW IS OPENSIFT DEDICATED SECURED?	8
1.3.1. TLS Support	8
CHAPTER 2. INFRASTRUCTURE COMPONENTS	11
2.1. KUBERNETES INFRASTRUCTURE	11
2.1.1. Overview	11
2.1.2. Masters	11
2.1.2.1. High Availability Masters	11
2.1.3. Nodes	12
2.1.3.1. Kubelet	12
2.1.3.2. Service Proxy	12
2.1.3.3. Node Object Definition	12
2.2. CONTAINER REGISTRY	13
2.2.1. Overview	13
2.2.2. Integrated OpenShift Container Registry	13
2.2.3. Third Party Registries	14
2.2.3.1. Authentication	14
2.3. WEB CONSOLE	14
2.3.1. Overview	14
2.3.2. CLI Downloads	14
2.3.3. Browser Requirements	15
2.3.4. Project Overviews	16
2.3.5. JVM Console	17
CHAPTER 3. CORE CONCEPTS	19
3.1. OVERVIEW	19
3.2. CONTAINERS AND IMAGES	19
3.2.1. Containers	19
3.2.2. Images	19
Image Version Tag Policy	20
3.2.3. Container Registries	20
3.3. PODS AND SERVICES	21
3.3.1. Pods	21
3.3.1.1. Pod Restart Policy	23
3.3.2. Services	24
3.3.2.1. Service Proxy	25
3.3.2.2. Headless services	25
3.3.2.2.1. Creating a headless service	26
3.3.2.2.2. Endpoint discovery by using a headless service	27
3.3.3. Labels	27
3.3.4. Endpoints	28
3.4. PROJECTS AND USERS	28
3.4.1. Users	28
3.4.2. Namespaces	29
3.4.3. Projects	29
3.4.3.1. Projects provided at installation	30
3.5. BUILDS AND IMAGE STREAMS	30
3.5.1. Builds	30
3.5.1.1. Docker Build	30

3.5.1.2. Source-to-Image (S2I) Build	31
3.5.1.3. Custom Build	31
3.5.1.4. Pipeline Build	32
3.5.2. Image Streams	32
3.5.2.1. Important terms	34
3.5.2.2. Configuring Image Streams	35
3.5.2.3. Image Stream Images	36
3.5.2.4. Image Stream Tags	36
3.5.2.5. Image Stream Change Triggers	38
3.5.2.6. Image Stream Mappings	38
3.5.2.7. Working with Image Streams	41
3.5.2.7.1. Getting Information about Image Streams	41
3.5.2.7.2. Adding Additional Tags to an Image Stream	42
3.5.2.7.3. Adding Tags for an External Image	43
3.5.2.7.4. Updating an Image Stream Tag	44
3.5.2.7.5. Removing Image Stream Tags from an Image Stream	44
3.5.2.7.6. Configuring Periodic Importing of Tags	44
3.6. DEPLOYMENTS	44
3.6.1. Replication controllers	45
3.6.2. Replica set	46
3.6.3. Jobs	47
3.6.4. Deployments and Deployment Configurations	47
3.7. TEMPLATES	48
3.7.1. Overview	48
CHAPTER 4. ADDITIONAL CONCEPTS	49
4.1. AUTHENTICATION	49
4.1.1. Overview	49
4.1.2. Users and Groups	49
4.1.3. API Authentication	49
4.1.3.1. Impersonation	50
4.1.4. OAuth	50
4.1.4.1. OAuth Clients	50
4.1.4.2. Service Accounts as OAuth Clients	51
4.1.4.3. Redirect URIs for Service Accounts as OAuth Clients	52
4.1.4.3.1. API Events for OAuth	54
4.1.4.3.1.1. Sample API Event Caused by a Possible Misconfiguration	55
4.1.4.4. Integrations	58
4.1.4.5. OAuth Server Metadata	58
4.1.4.6. Obtaining OAuth Tokens	59
4.2. AUTHORIZATION	60
4.2.1. Overview	60
4.2.2. Evaluating Authorization	61
4.2.3. Cluster and Local RBAC	62
4.2.4. Cluster Roles and Local Roles	62
4.2.5. Security Context Constraints	63
4.2.5.1. SCC Strategies	67
4.2.5.1.1. RunAsUser	67
4.2.5.1.2. SELinuxContext	67
4.2.5.1.3. SupplementalGroups	67
4.2.5.1.4. FSGroup	67
4.2.5.2. Controlling Volumes	68
4.2.5.3. Restricting Access to FlexVolumes	69

4.2.5.4. Seccomp	69
4.2.5.5. Admission	69
4.2.5.5.1. SCC Prioritization	70
4.2.5.5.2. Understanding Pre-allocated Values and Security Context Constraints	71
4.3. PERSISTENT STORAGE	72
4.3.1. Overview	72
4.3.2. Lifecycle of a Volume and Claim	72
4.3.2.1. Provisioning	72
4.3.2.2. Binding	72
4.3.2.3. Using	73
4.3.2.4. Releasing	73
4.3.2.5. Reclaiming	73
4.3.2.5.1. Recycling	73
4.3.3. Persistent Volumes	74
4.3.3.1. Types of Persistent Volumes	74
4.3.3.2. Capacity	75
4.3.3.3. Access Modes	75
4.3.3.4. OpenShift Dedicated Restrictions	76
4.3.3.5. Reclaim Policy	77
4.3.3.6. Phase	77
4.3.4. Persistent Volume Claims	78
4.3.4.1. Storage Class	78
4.3.4.2. Access Modes	78
4.3.4.3. Resources	78
4.3.4.4. Claims As Volumes	78
4.4. SOURCE CONTROL MANAGEMENT	79
4.5. ADMISSION CONTROLLERS	79
4.5.1. Overview	79
4.5.2. General Admission Rules	80
4.6. OTHER API OBJECTS	81
4.6.1. LimitRange	81
4.6.2. ResourceQuota	81
4.6.3. Resource	81
4.6.4. Secret	81
4.6.5. PersistentVolume	81
4.6.6. PersistentVolumeClaim	81
4.6.6.1. Custom Resources	82
4.6.7. OAuth Objects	82
4.6.7.1. OAuthClient	82
4.6.7.2. OAuthClientAuthorization	83
4.6.7.3. OAuthAuthorizeToken	83
4.6.7.4. OAuthAccessToken	84
4.6.8. User Objects	85
4.6.8.1. Identity	85
4.6.8.2. User	86
4.6.8.3. UserIdentityMapping	86
4.6.8.4. Group	87
CHAPTER 5. NETWORKING	88
5.1. NETWORKING	88
5.1.1. Overview	88
5.1.2. OpenShift Dedicated DNS	88
5.2. HAPROXY ROUTER PLUG-IN	89

5.2.1. Overview	89
5.2.2. HAProxy Template Router	89
5.2.2.1. HAProxy Template Router Metrics	90
5.3. ROUTES	93
5.3.1. Overview	93
5.3.2. Route-specific Annotations	93
5.3.3. Route-specific IP Whitelists	95

CHAPTER 1. OVERVIEW

OpenShift v3 is a layered system designed to expose underlying Docker-formatted container image and Kubernetes concepts as accurately as possible, with a focus on easy composition of applications by a developer. For example, install Ruby, push code, and add MySQL.

Unlike OpenShift v2, more flexibility of configuration is exposed after creation in all aspects of the model. The concept of an application as a separate object is removed in favor of more flexible composition of "services", allowing two web containers to reuse a database or expose a database directly to the edge of the network.

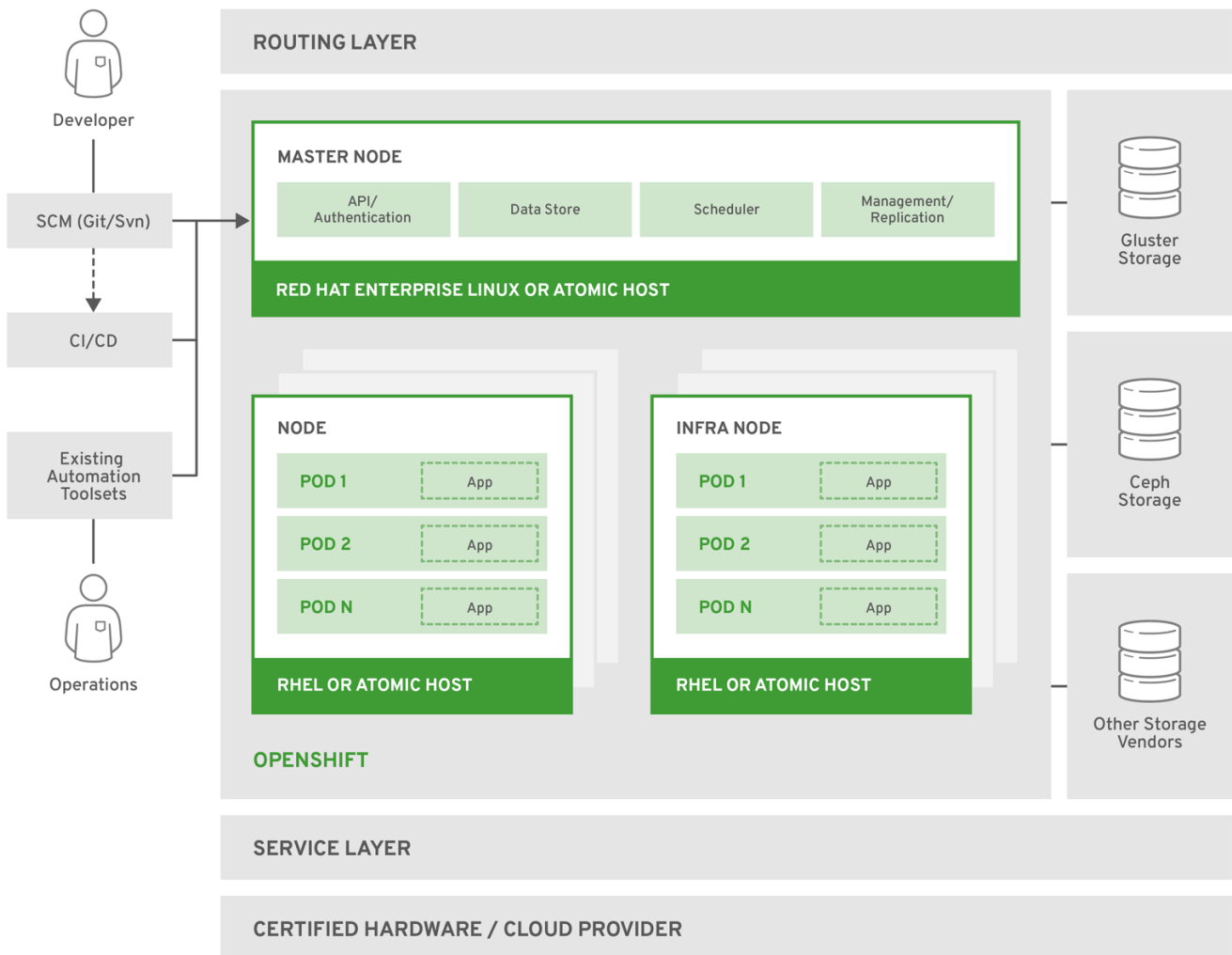
1.1. WHAT ARE THE LAYERS?

The Docker service provides the abstraction for packaging and creating Linux-based, lightweight [container images](#). Kubernetes provides the [cluster management](#) and orchestrates containers on multiple hosts.

OpenShift Dedicated adds:

- Source code management, [builds](#), and [deployments](#) for developers
- Managing and promoting [images](#) at scale as they flow through your system
- Application management at scale
- Team and user tracking for organizing a large developer organization
- Networking infrastructure that supports the cluster

Figure 1.1. OpenShift Dedicated Architecture Overview



OPENSIFT_415489_0218

1.2. WHAT IS THE OPENSIFT DEDICATED ARCHITECTURE?

OpenShift Dedicated 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](#).
- 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 OpenShift Dedicated 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 behaviors 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 take action. OpenShift Dedicated makes the ability to customize the cluster in this way a first-class behavior.

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, then 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.

1.3. HOW IS OPENSIFT DEDICATED SECURED?

The OpenShift Dedicated and Kubernetes APIs [authenticate](#) users who present credentials, and then [authorize](#) them based on their role. Both developers and administrators can be authenticated via a number of means, primarily [OAuth tokens](#) and X.509 client certificates. OAuth tokens are signed with JSON Web Algorithm *RS256*, which is RSA signature algorithm PKCS#1 v1.5 with SHA-256.

Developers (clients of the system) typically make REST API calls from a [client program](#) like `oc` or to the [web console](#) via their browser, and use OAuth bearer tokens for most communications. Infrastructure components (like nodes) use client certificates generated by the system that contain their identities. Infrastructure components that run in containers use a token associated with their [service account](#) to connect to the API.

Authorization is handled in the OpenShift Dedicated policy engine, which defines actions like "create pod" or "list services" and groups them into roles in a policy document. Roles are bound to users or groups by the user or group identifier. When a user or service account attempts an action, the policy engine checks for one or more of the roles assigned to the user (e.g., cluster administrator or administrator of the current project) before allowing it to continue.

1.3.1. TLS Support

All communication channels with the REST API, as well as between [master components](#) such as etcd and the API server, are secured with TLS. TLS provides strong encryption, data integrity, and authentication of servers with X.509 server certificates and public key infrastructure. By default, a new internal PKI is created for each deployment of OpenShift Dedicated. The internal PKI uses 2048 bit RSA keys and SHA-256 signatures.

OpenShift Dedicated uses Golang's standard library implementation of [crypto/tls](#) and does not depend on any external crypto and TLS libraries. Additionally, the client depends on external libraries for GSSAPI authentication and OpenPGP signatures. GSSAPI is typically provided by either MIT Kerberos or Heimdal Kerberos, which both use OpenSSL's libcrypto. OpenPGP signature verification is handled by libpgme and GnuPG.

The insecure versions SSL 2.0 and SSL 3.0 are unsupported and not available. The OpenShift Dedicated server and `oc` client only provide TLS 1.2 by default. TLS 1.0 and TLS 1.1 can be enabled in the server configuration. Both server and client prefer modern cipher suites with authenticated encryption algorithms and perfect forward secrecy. Cipher suites with deprecated and insecure algorithms such as RC4, 3DES, and MD5 are disabled. Some internal clients (for example, LDAP authentication) have less restrict settings with TLS 1.0 to 1.2 and more cipher suites enabled.

Table 1.1. Supported TLS Versions

TLS Version	OpenShift Dedicated Server	oc Client	Other Clients
SSL 2.0	Unsupported	Unsupported	Unsupported
SSL 3.0	Unsupported	Unsupported	Unsupported
TLS 1.0	No ^[a]	No ^[a]	Maybe ^[b]
TLS 1.1	No ^[a]	No ^[a]	Maybe ^[b]
TLS 1.2	Yes	Yes	Yes
TLS 1.3	N/A ^[c]	N/A ^[c]	N/A ^[c]
<p>^[a] Disabled by default, but can be enabled in the server configuration.</p> <p>^[b] Some internal clients, such as the LDAP client.</p> <p>^[c] TLS 1.3 is still under development.</p>			

The following list of enabled cipher suites of OpenShift Dedicated's server and **oc** client are sorted in preferred order:

- **TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305**
- **TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305**
- **TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256**
- **TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256**
- **TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384**
- **TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384**
- **TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256**
- **TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256**
- **TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA**
- **TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA**
- **TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA**
- **TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA**
- **TLS_RSA_WITH_AES_128_GCM_SHA256**
- **TLS_RSA_WITH_AES_256_GCM_SHA384**

- `TLS_RSA_WITH_AES_128_CBC_SHA`
- `TLS_RSA_WITH_AES_256_CBC_SHA`

CHAPTER 2. INFRASTRUCTURE COMPONENTS

2.1. KUBERNETES INFRASTRUCTURE

2.1.1. Overview

Within OpenShift Dedicated, Kubernetes manages containerized applications across a set of containers or hosts and provides mechanisms for deployment, maintenance, and application-scaling. The Docker service packages, instantiates, and runs containerized applications.

A Kubernetes cluster consists of one or more masters and a set of nodes. You can optionally configure your masters for [high availability](#) (HA) to ensure that the cluster has no single point of failure.



NOTE

OpenShift Dedicated uses Kubernetes 1.7 and Docker 1.12.

2.1.2. Masters

The master is the host or hosts that contain the master components, including the API server, controller manager server, and **etcd**. The master manages [nodes](#) in its Kubernetes cluster and schedules [pods](#) to run on nodes.

Table 2.1. Master Components

Component	Description
API Server	The Kubernetes API server validates and configures the data for pods, services, and replication controllers. It also assigns pods to nodes and synchronizes pod information with service configuration. Can be run as a standalone process.
etcd	etcd stores the persistent master state while other components watch etcd for changes to bring themselves into the desired state. etcd can be optionally configured for high availability, typically deployed with 2n+1 peer services.
Controller Manager Server	The controller manager server watches etcd for changes to replication controller objects and then uses the API to enforce the desired state. Can be run as a standalone process. Several such processes create a cluster with one active leader at a time.
HAProxy	Optional, used when configuring highly-available masters with the native method to balance load between API master endpoints.

2.1.2.1. High Availability Masters

The availability of running applications remains if the master or any of its services fail. However, failure of master services reduces the ability of the system to respond to application failures or creation of new applications.

OpenShift Dedicated's advanced installation method (see the [OpenShift Enterprise Cluster Administration](#) documentation for details) provides specific examples using the **native** HA method and configuring HAProxy. You can also take the concepts and apply them towards your existing HA solutions

using the **native** method instead of HAProxy.

When using the **native** HA method with HAProxy, master components have the following availability:

Table 2.2. Availability Matrix with HAProxy

Role	Style	Notes
etcd	Active-active	Fully redundant deployment with load balancing
API Server	Active-active	Managed by HAProxy
Controller Manager Server	Active-passive	One instance is elected as a cluster leader at a time
HAProxy	Active-passive	Balances load between API master endpoints

2.1.3. Nodes

A node provides the runtime environments for containers. Each node in a Kubernetes cluster has the required services to be managed by the [master](#). Nodes also have the required services to run pods, including the Docker service, a [kubelet](#), and a [service proxy](#).

OpenShift Dedicated creates nodes from a cloud provider, physical systems, or virtual systems. Kubernetes interacts with [node objects](#) that are a representation of those nodes. The master uses the information from node objects to validate nodes with health checks. A node is ignored until it passes the health checks, and the master continues checking nodes until they are valid. The [Kubernetes documentation](#) has more information on node management.

2.1.3.1. Kubelet

Each node has a kubelet that updates the node as specified by a container manifest, which is a YAML file that describes a pod. The kubelet uses a set of manifests to ensure that its containers are started and that they continue to run.

A container manifest can be provided to a kubelet by:

- A file path on the command line that is checked every 20 seconds.
- An HTTP endpoint passed on the command line that is checked every 20 seconds.
- The kubelet watching an **etcd** server, such as `/registry/hosts/${hostname -f}`, and acting on any changes.
- The kubelet listening for HTTP and responding to a simple API to submit a new manifest.

2.1.3.2. Service Proxy

Each node also runs a simple network proxy that reflects the services defined in the API on that node. This allows the node to do simple TCP and UDP stream forwarding across a set of back ends.

2.1.3.3. Node Object Definition

The following is an example node object definition in Kubernetes:

```
apiVersion: v1 ❶
kind: Node ❷
metadata:
  creationTimestamp: null
  labels: ❸
    kubernetes.io/hostname: node1.example.com
  name: node1.example.com ❹
spec:
  externalID: node1.example.com ❺
status:
  nodeInfo:
    bootID: ""
    containerRuntimeVersion: ""
    kernelVersion: ""
    kubeProxyVersion: ""
    kubeletVersion: ""
    machineID: ""
    osImage: ""
    systemUUID: ""
```

- ❶ **apiVersion** defines the API version to use.
- ❷ **kind** set to **Node** identifies this as a definition for a node object.
- ❸ **metadata.labels** lists any [labels](#) that have been added to the node.
- ❹ **metadata.name** is a required value that defines the name of the node object. This value is shown in the **NAME** column when running the **oc get nodes** command.
- ❺ **spec.externalID** defines the fully-qualified domain name where the node can be reached. Defaults to the **metadata.name** value when empty.

2.2. CONTAINER REGISTRY

2.2.1. Overview

OpenShift Dedicated can utilize any server implementing the Docker registry API as a source of images, including the Docker Hub, private registries run by third parties, and the integrated OpenShift Dedicated registry.

2.2.2. Integrated OpenShift Container Registry

OpenShift Dedicated provides an integrated container registry called *OpenShift Container Registry* (OCR) that adds the ability to automatically provision new image repositories on demand. This provides users with a built-in location for their application [builds](#) to push the resulting images.

Whenever a new image is pushed to OCR, the registry notifies OpenShift Dedicated about the new image, passing along all the information about it, such as the namespace, name, and image metadata. Different pieces of OpenShift Dedicated react to new images, creating new [builds](#) and [deployments](#).

2.2.3. Third Party Registries

OpenShift Dedicated can create containers using images from third party registries, but it is unlikely that these registries offer the same image notification support as the integrated OpenShift Dedicated registry. In this situation OpenShift Dedicated will fetch tags from the remote registry upon imagestream creation. Refreshing the fetched tags is as simple as running `oc import-image <stream>`. When new images are detected, the previously-described build and deployment reactions occur.

2.2.3.1. Authentication

OpenShift Dedicated can communicate with registries to access private image repositories using credentials supplied by the user. This allows OpenShift to push and pull images to and from private repositories. The [Authentication](#) topic has more information.

2.3. WEB CONSOLE

2.3.1. Overview

The OpenShift Dedicated web console is a user interface accessible from a web browser. Developers can use the web console to visualize, browse, and manage the contents of [projects](#).

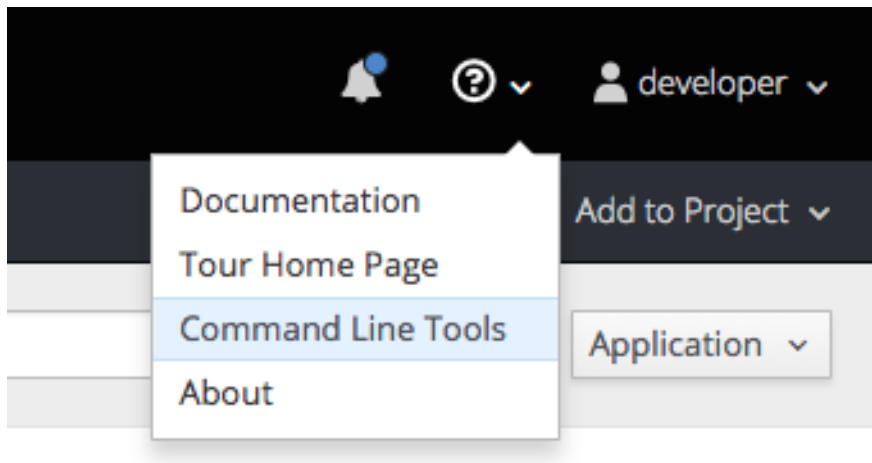


NOTE

JavaScript must be enabled to use the web console. For the best experience, use a web browser that supports [WebSockets](#).

2.3.2. CLI Downloads

You can access CLI downloads from the Help icon in the web console:



Command Line Tools

With the OpenShift command line interface (CLI), you can create applications and manage OpenShift projects from a terminal. You can download the `oc` client tool using the links below. For more information about downloading and installing it, please refer to the [Get Started with the CLI](#) documentation.

Download `oc` :

[Latest Release](#) 

After downloading and installing it, you can start by logging in. You are currently logged into this console as **developer**. If you want to log into the CLI using the same session token:

```
oc login https://127.0.0.1:8443 --token=<hidden>
```



A token is a form of a password. Do not share your API token. To reveal your token, press the copy to clipboard button and then paste the clipboard contents.

After you login to your account you will get a list of projects that you can switch between:

```
oc project <project-name>
```

If you do not have any existing projects, you can create one:

```
oc new-project <project-name>
```

To show a high level overview of the current project:

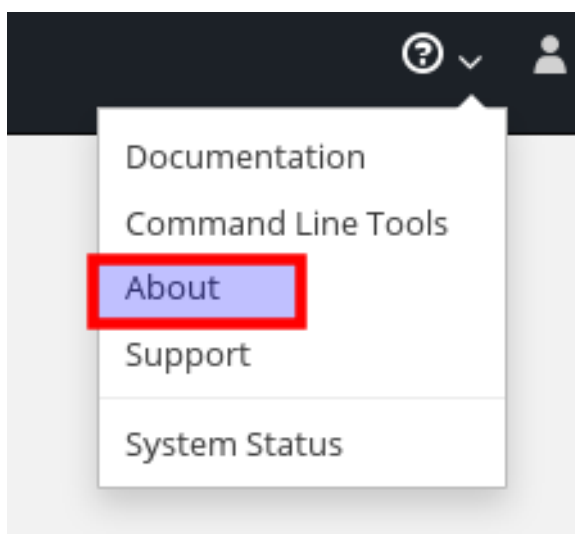
```
oc status
```

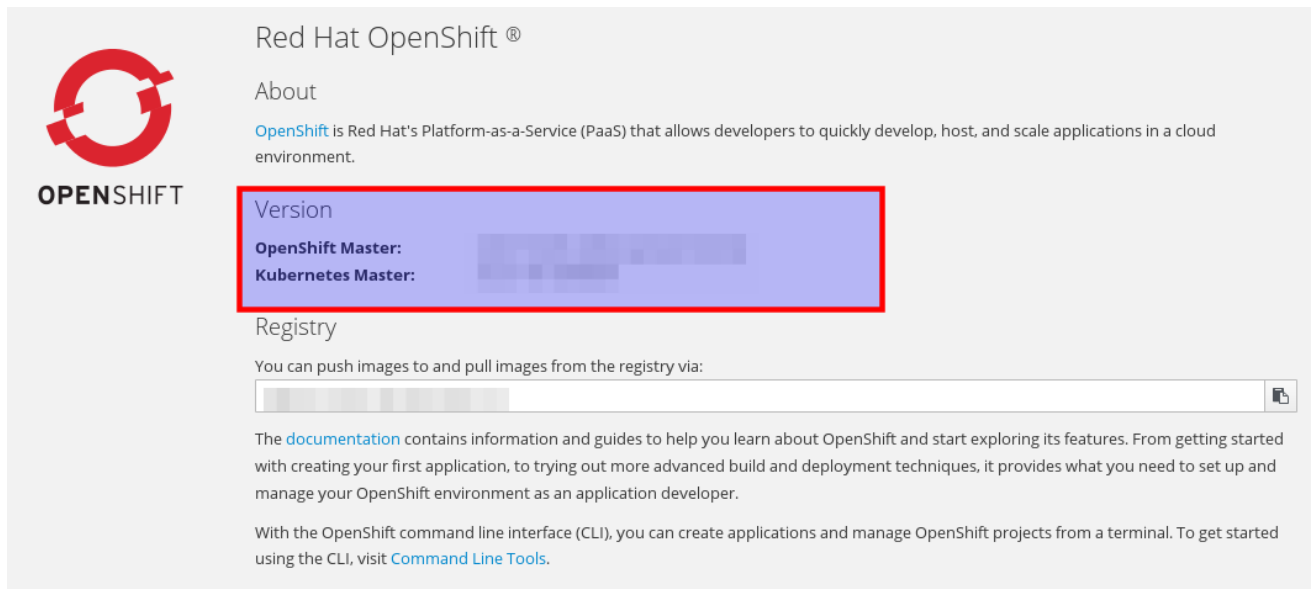
For other information about the command line tools, check the [CLI Reference](#) and [Basic CLI Operations](#).

2.3.3. Browser Requirements

Review the [tested integrations](#) for OpenShift Dedicated.

From the **About** page in the web console, you can check the cluster's version number.

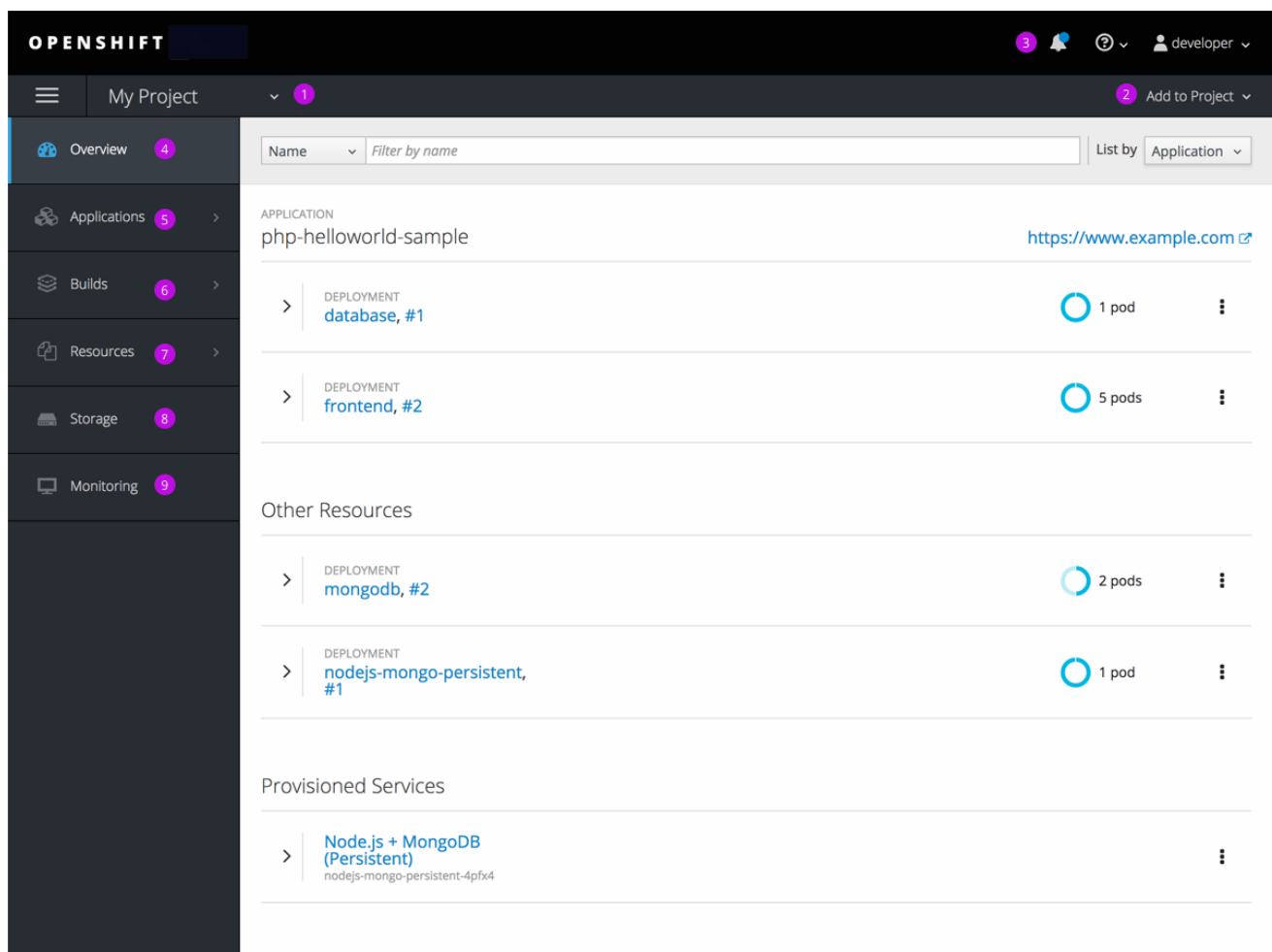




2.3.4. Project Overviews

After [logging in](#), the web console provides developers with an overview for the currently selected [project](#):

Figure 2.1. Web Console Project Overview



The project selector allows you to [switch between projects](#) you have access to.

Create new applications [using a source repository](#) or service from the service catalog.

Notifications related to your project.

The **Overview** tab (currently selected) visualizes the contents of your project with a high-level view of each component.

Applications tab: Browse and perform actions on your deployments, pods, services, and routes.

Builds tab: Browse and perform actions on your builds and image streams.

Resources tab: View your current quota consumption and other resources.

Storage tab: View persistent volume claims and request storage for your applications.

Monitoring tab: View logs for builds, pods, and deployments, as well as event notifications for all objects in your project.






2.3.5. JVM Console

For pods based on Java images, the web console also exposes access to a hawt.io-based JVM console for viewing and managing any relevant integration components. A **Connect** link is displayed in the pod's details on the *Browse* → *Pods* page, provided the container has a port named **jolokia**.

Figure 2.2. Pod with a Link to the JVM Console

Template

CONTAINER: STI-BUILD

-  **Image:** openshift/origin-sti-builder:latest
-  **Mount:** docker-socket → /var/run/docker.sock
-  **Mount:** builder-dockercfg-p7gmj-push → /var/run/secrets/openshift.io/push
-  **Mount:** builder-token-t6b9i → /var/run/secrets/kubernetes.io/serviceaccount
-  [Open Java Console](#)

Volumes

docker-socket

Type:	host path (bare host directory volume)
Path:	/var/run/docker.sock

After connecting to the JVM console, different pages are displayed depending on which components are relevant to the connected pod.

Figure 2.3. JVM Console

Connected to quickstart-java-camel-spring-container

[← Back](#)

JMX Threads Camel

Total: 9 Runnable: 3 Timed waiting: 2 Waiting: 4

Filter... ✕

ID	State	Name	Waited Time	Blocked Time	Native	Suspended
15		Thread-5	1 hour			
14		Camel (camel-1) thread #0 - file://src/data	1 hour			
9		Jolokia Agent Cleanup Thread				
8		Thread-3		279 ms	(in native)	
6		server-timer	1 hour			
4		Signal Dispatcher				
3		Finalizer	1 hour			
2		Reference Handler	1 hour	10 ms		
1		main				

The following pages are available:

Page	Description
JMX	View and manage JMX domains and mbeans.
Threads	View and monitor the state of threads.
ActiveMQ	View and manage Apache ActiveMQ brokers.
Camel	View and and manage Apache Camel routes and dependencies.
OSGi	View and manage the JBoss Fuse OSGi environment.

CHAPTER 3. CORE CONCEPTS

3.1. OVERVIEW

The following topics provide high-level, architectural information on core concepts and objects you will encounter when using OpenShift Dedicated. Many of these objects come from Kubernetes, which is extended by OpenShift Dedicated to provide a more feature-rich development lifecycle platform.

- [Containers and images](#) are the building blocks for deploying your applications.
- [Pods and services](#) allow for containers to communicate with each other and proxy connections.
- [Projects and users](#) provide the space and means for communities to organize and manage their content together.
- [Builds and image streams](#) allow you to build working images and react to new images.
- [Deployments](#) add expanded support for the software development and deployment lifecycle.
- [Routes](#) announce your service to the world.
- [Templates](#) allow for many objects to be created at once based on customized parameters.

3.2. CONTAINERS AND IMAGES

3.2.1. Containers

The basic units of OpenShift Dedicated applications are called *containers*. [Linux container technologies](#) are lightweight mechanisms for isolating running processes so that they are limited to interacting with only their designated resources.

Many application instances can be running in containers on a single host without visibility into each others' processes, files, network, and so on. Typically, each container provides a single service (often called a "micro-service"), such as a web server or a database, though containers can be used for arbitrary workloads.

The Linux kernel has been incorporating capabilities for container technologies for years. More recently the Docker project has developed a convenient management interface for Linux containers on a host. OpenShift Dedicated and Kubernetes add the ability to orchestrate Docker-formatted containers across multi-host installations.

Though you do not directly interact with the Docker CLI or service when using OpenShift Dedicated, understanding their capabilities and terminology is important for understanding their role in OpenShift Dedicated and how your applications function inside of containers. The **docker** RPM is available as part of RHEL 7, as well as CentOS and Fedora, so you can experiment with it separately from OpenShift Dedicated. Refer to the article [Get Started with Docker Formatted Container Images on Red Hat Systems](#) for a guided introduction.

3.2.2. Images

Containers in OpenShift Dedicated are based on Docker-formatted container *images*. An image is a binary that includes all of the requirements for running a single container, as well as metadata describing its needs and capabilities.

You can think of it as a packaging technology. Containers only have access to resources defined in the image unless you give the container additional access when creating it. By deploying the same image in multiple containers across multiple hosts and load balancing between them, OpenShift Dedicated can provide redundancy and horizontal scaling for a service packaged into an image.

You can use the Docker CLI directly to build images, but OpenShift Dedicated also supplies builder images that assist with creating new images by adding your code or configuration to existing images.

Because applications develop over time, a single image name can actually refer to many different versions of the "same" image. Each different image is referred to uniquely by its hash (a long hexadecimal number e.g. **fd44297e2ddb050ec4f...**) which is usually shortened to 12 characters (e.g. **fd44297e2ddb**).

Image Version Tag Policy

Rather than version numbers, the Docker service allows applying tags (such as **v1**, **v2.1**, **GA**, or the default **latest**) in addition to the image name to further specify the image desired, so you may see the same image referred to as **centos** (implying the **latest** tag), **centos:centos7**, or **fd44297e2ddb**.



WARNING

Do not use the **latest** tag for any official OpenShift Dedicated images. These are images that start with **openshift3/**. **latest** can refer to a number of versions, such as **3.4**, or **3.5**.

How you tag the images dictates the updating policy. The more specific you are, the less frequently the image will be updated. Use the following to determine your chosen OpenShift Dedicated images policy:

vX.Y

The vX.Y tag points to X.Y.Z-<number>. For example, if the **registry-console** image is updated to v3.4, it points to the newest 3.4.Z-<number> tag, such as 3.4.1-8.

X.Y.Z

Similar to the vX.Y example above, the X.Y.Z tag points to the latest X.Y.Z-<number>. For example, 3.4.1 would point to 3.4.1-8

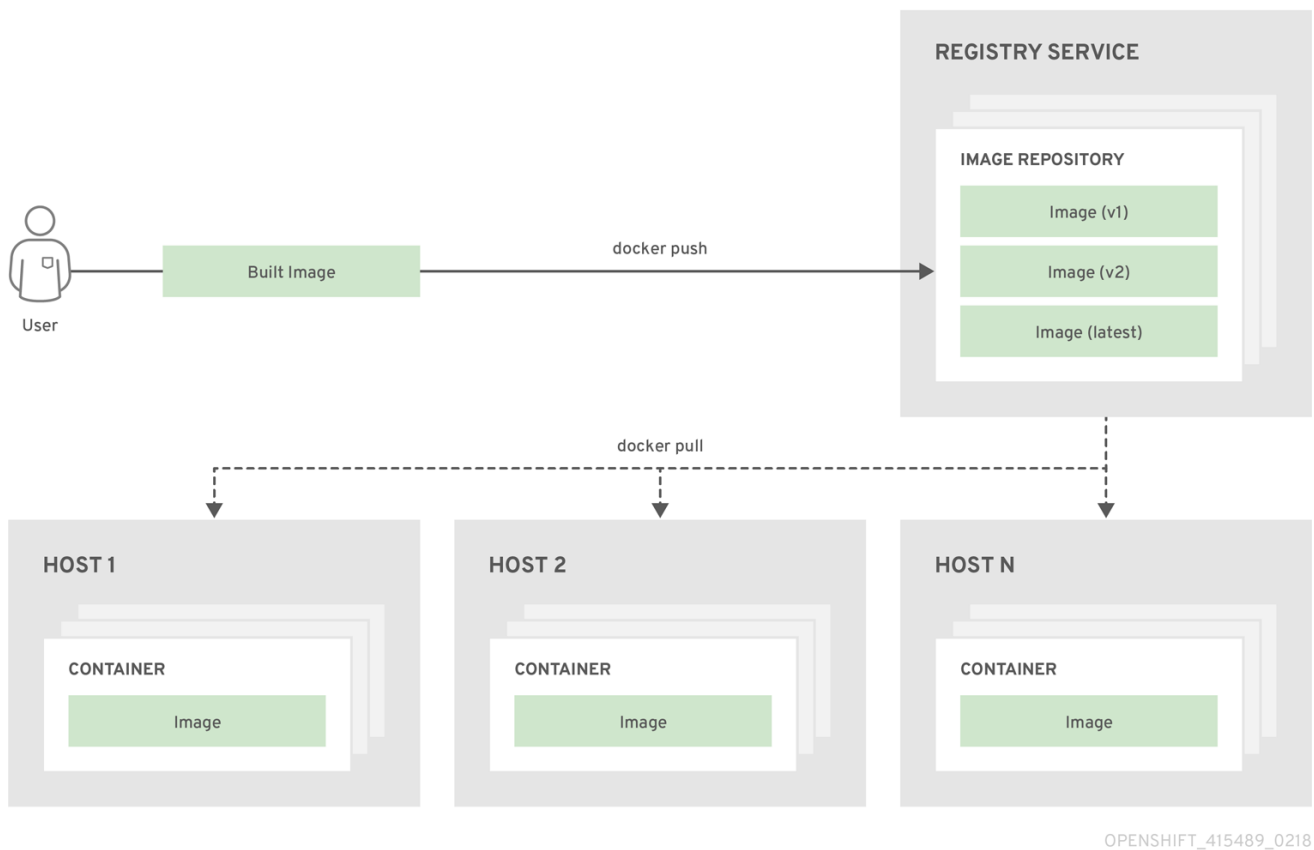
X.Y.Z-<number>

The tag is unique and does not change. When using this tag, the image does not update if an image is updated. For example, the 3.4.1-8 will always point to 3.4.1-8, even if an image is updated.

3.2.3. Container Registries

A container registry is a service for storing and retrieving Docker-formatted container images. A registry contains a collection of one or more image repositories. Each image repository contains one or more tagged images. Docker provides its own registry, the [Docker Hub](#), and you can also use private or third-party registries. Red Hat provides a registry at **registry.access.redhat.com** for subscribers. OpenShift Dedicated can also supply its own internal registry for managing custom container images.

The relationship between containers, images, and registries is depicted in the following diagram:



3.3. PODS AND SERVICES

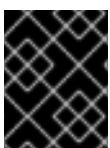
3.3.1. Pods

OpenShift Dedicated leverages the Kubernetes concept of a *pod*, which is one or more [containers](#) deployed together on one host, and the smallest compute unit that can be defined, deployed, and managed.

Pods are the rough equivalent of a machine instance (physical or virtual) to a container. Each pod is allocated its own internal IP address, therefore owning its entire port space, and containers within pods can share their local storage and networking.

Pods have a lifecycle; they are defined, then they are assigned to run on a node, then they run until their container(s) exit or they are removed for some other reason. Pods, depending on policy and exit code, may be removed after exiting, or may be retained in order to enable access to the logs of their containers.

OpenShift Dedicated treats pods as largely immutable; changes cannot be made to a pod definition while it is running. OpenShift Dedicated implements changes by terminating an existing pod and recreating it with modified configuration, base image(s), or both. Pods are also treated as expendable, and do not maintain state when recreated. Therefore pods should usually be managed by higher-level [controllers](#), rather than directly by users.



IMPORTANT

The recommended maximum number of pods per OpenShift Dedicated node host is 35. You can have no more than 40 pods per node.

**WARNING**

Bare pods that are not managed by a [replication controller](#) will be not rescheduled upon node disruption.

Below is an example definition of a pod that provides a long-running service, which is actually a part of the OpenShift Dedicated infrastructure: the integrated container registry. It demonstrates many features of pods, most of which are discussed in other topics and thus only briefly mentioned here:

Example 3.1. Pod Object Definition (YAML)

```

apiVersion: v1
kind: Pod
metadata:
  annotations: { ... }
  labels:
    deployment: docker-registry-1
    deploymentconfig: docker-registry
    docker-registry: default
  generateName: docker-registry-1-
spec:
  containers:
  - env:
    - name: OPENSIFT_CA_DATA
      value: ...
    - name: OPENSIFT_CERT_DATA
      value: ...
    - name: OPENSIFT_INSECURE
      value: "false"
    - name: OPENSIFT_KEY_DATA
      value: ...
    - name: OPENSIFT_MASTER
      value: https://master.example.com:8443
    image: openshift/origin-docker-registry:v0.6.2
    imagePullPolicy: IfNotPresent
    name: registry
    ports:
    - containerPort: 5000
      protocol: TCP
    resources: {}
    securityContext: { ... }
    volumeMounts:
    - mountPath: /registry
      name: registry-storage
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: default-token-br6yz
      readOnly: true
  dnsPolicy: ClusterFirst
  imagePullSecrets:
  - name: default-dockercfg-at06w
  
```

```

restartPolicy: Always
serviceAccount: default
volumes:
- emptyDir: {}
  name: registry-storage
- name: default-token-br6yz
  secret:
    secretName: default-token-br6yz

```

9
10
11

- 1 Pods can be "tagged" with one or more [labels](#), which can then be used to select and manage groups of pods in a single operation. The labels are stored in key/value format in the **metadata** hash. One label in this example is **docker-registry=default**.
- 2 Pods must have a unique name within their [namespace](#). A pod definition may specify the basis of a name with the **generateName** attribute, and random characters will be added automatically to generate a unique name.
- 3 **containers** specifies an array of container definitions; in this case (as with most), just one.
- 4 Environment variables can be specified to pass necessary values to each container.
- 5 Each container in the pod is instantiated from its own [Docker-formatted container image](#).
- 6 The container can bind to ports which will be made available on the pod's IP.
- 7 OpenShift Dedicated defines a [security context](#) for containers which specifies whether they are allowed to run as privileged containers, run as a user of their choice, and more. The default context is very restrictive but administrators can modify this as needed.
- 8 The container specifies where external storage volumes should be mounted within the container. In this case, there is a volume for storing the registry's data, and one for access to credentials the registry needs for making requests against the OpenShift Dedicated API.
- 9 The [pod restart policy](#) with possible values **Always**, **OnFailure**, and **Never**. The default value is **Always**.
- 10 Pods making requests against the OpenShift Dedicated API is a common enough pattern that there is a **serviceAccount** field for specifying which [service account](#) user the pod should authenticate as when making the requests. This enables fine-grained access control for custom infrastructure components.
- 11 The pod defines storage volumes that are available to its container(s) to use. In this case, it provides an ephemeral volume for the registry storage and a **secret** volume containing the service account credentials.



NOTE

This pod definition does not include attributes that are filled by OpenShift Dedicated automatically after the pod is created and its lifecycle begins. The [Kubernetes pod documentation](#) has details about the functionality and purpose of pods.

3.3.1.1. Pod Restart Policy

A pod restart policy determines how OpenShift Dedicated responds when containers in that pod exit. The policy applies to all containers in that pod.

The possible values are:

- **Always** - Tries restarting a successfully exited container on the pod continuously, with an exponential back-off delay (10s, 20s, 40s) until the pod is restarted. The default is **Always**.
- **OnFailure** - Tries restarting a failed container on the pod with an exponential back-off delay (10s, 20s, 40s) capped at 5 minutes.
- **Never** - Does not try to restart exited or failed containers on the pod. Pods immediately fail and exit.

Once bound to a node, a pod will never be bound to another node. This means that a controller is necessary in order for a pod to survive node failure:

Condition	Controller Type	Restart Policy
Pods that are expected to terminate (such as batch computations)	Job	OnFailure or Never
Pods that are expected to not terminate (such as web servers)	Replication Controller	Always .
Pods that need to run one-per-machine	Daemonset	Any

If a container on a pod fails and the restart policy is set to **OnFailure**, the pod stays on the node and the container is restarted. If you do not want the container to restart, use a restart policy of **Never**.

If an entire pod fails, OpenShift Dedicated starts a new pod. Developers need to address the possibility that applications might be restarted in a new pod. In particular, applications need to handle temporary files, locks, incomplete output, and so forth caused by previous runs.

For details on how OpenShift Dedicated uses restart policy with failed containers, see the [Example States](#) in the Kubernetes documentation.

3.3.2. Services

A Kubernetes [service](#) serves as an internal load balancer. It identifies a set of replicated [pods](#) in order to proxy the connections it receives to them. Backing pods can be added to or removed from a service arbitrarily while the service remains consistently available, enabling anything that depends on the service to refer to it at a consistent address. The default service clusterIP addresses are from the OpenShift Dedicated internal network and they are used to permit pods to access each other.

Services are assigned an IP address and port pair that, when accessed, proxy to an appropriate backing pod. A service uses a label selector to find all the containers running that provide a certain network service on a certain port.

Like pods, services are REST objects. The following example shows the definition of a service for the pod defined above:

Example 3.2. Service Object Definition (YAML)

```

apiVersion: v1
kind: Service
metadata:
  name: docker-registry
spec:
  selector:
    docker-registry: default
  clusterIP: 172.30.136.123
  ports:
  - nodePort: 0
    port: 5000
    protocol: TCP
    targetPort: 5000

```

- 1 The service name **docker-registry** is also used to construct an environment variable with the service IP that is inserted into other pods in the same namespace. The maximum name length is 63 characters.
- 2 The label selector identifies all pods with the **docker-registry=default** label attached as its backing pods.
- 3 Virtual IP of the service, allocated automatically at creation from a pool of internal IPs.
- 4 Port the service listens on.
- 5 Port on the backing pods to which the service forwards connections.

The [Kubernetes documentation](#) has more information on services.

3.3.2.1. Service Proxy

OpenShift Dedicated has an **iptables**-based implementation of the service-routing infrastructure. It uses probabilistic **iptables** rewriting rules to distribute incoming service connections between the endpoint pods. It also requires that all endpoints are always able to accept connections.

3.3.2.2. Headless services

If your application does not need load balancing or single-service IP addresses, you can create a headless service. When you create a headless service, no load-balancing or proxying is done and no cluster IP is allocated for this service. For such services, DNS is automatically configured depending on whether the service has selectors defined or not.

Services with selectors: For headless services that define selectors, the endpoints controller creates **Endpoints** records in the API and modifies the DNS configuration to return **A** records (addresses) that point directly to the pods backing the service.

Services without selectors: For headless services that do not define selectors, the endpoints controller does not create **Endpoints** records. However, the DNS system looks for and configures the following records:

- For **ExternalName** type services, **CNAME** records.
- For all other service types, **A** records for any endpoints that share a name with the service.

3.3.2.2.1. Creating a headless service

Creating a headless service is similar to creating a standard service, but you do not declare the **ClusterIP** address. To create a headless service, add the **clusterIP: None** parameter value to the service YAML definition.

For example, for a group of pods that you want to be a part of the same cluster or service.

List of pods

```
$ oc get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
frontend-1-287hw	1/1	Running	0	7m	172.17.0.3	
node_1						
frontend-1-68km5	1/1	Running	0	7m	172.17.0.6	
node_1						

You can define the headless service as:

Headless service definition

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: ruby-helloworld-sample
    template: application-template-stibuild
  name: frontend-headless ❶
spec:
  clusterIP: None ❷
  ports:
  - name: web
    port: 5432
    protocol: TCP
    targetPort: 8080
  selector:
    name: frontend ❸
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

- ❶ Name of the headless service.
- ❷ Setting **clusterIP** variable to **None** declares a headless service.
- ❸ Selects all pods that have **frontend** label.

Also, headless service does not have any IP address of its own.

■

```
$ oc get svc
NAME                                TYPE             CLUSTER-IP      EXTERNAL-IP      PORT(S)
AGE
frontend                           ClusterIP         172.30.232.77   <none>           5432/TCP
12m
frontend-headless                   ClusterIP         None            <none>           5432/TCP
10m
```

3.3.2.2.2. Endpoint discovery by using a headless service

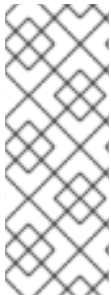
The benefit of using a headless service is that you can discover a pod's IP address directly. Standard services act as load balancer or proxy and give access to the workload object by using the service name. With headless services, the service name resolves to the set of IP addresses of the pods that are grouped by the service.

When you look up the DNS **A** record for a standard service, you get the loadbalanced IP of the service.

```
$ dig frontend.test A +search +short
172.30.232.77
```

But for a headless service, you get the list of IPs of individual pods.

```
$ dig frontend-headless.test A +search +short
172.17.0.3
172.17.0.6
```



NOTE

For using a headless service with a StatefulSet and related use cases where you need to resolve DNS for the pod during initialization and termination, set **publishNotReadyAddresses** to **true** (the default value is **false**). When **publishNotReadyAddresses** is set to **true**, it indicates that DNS implementations must publish the **notReadyAddresses** of subsets for the Endpoints associated with the Service.

3.3.3. Labels

Labels are used to organize, group, or select API objects. For example, [pods](#) are "tagged" with labels, and then [services](#) use label selectors to identify the pods they proxy to. This makes it possible for services to reference groups of pods, even treating pods with potentially different containers as related entities.

Most objects can include labels in their metadata. So labels can be used to group arbitrarily-related objects; for example, all of the [pods](#), [services](#), [replication controllers](#), and [deployment configurations](#) of a particular application can be grouped.

Labels are simple key/value pairs, as in the following example:

```
labels:
  key1: value1
  key2: value2
```

Consider:

- A pod consisting of an **nginx** container, with the label **role=webserver**.
- A pod consisting of an **Apache httpd** container, with the same label **role=webserver**.

A service or replication controller that is defined to use pods with the **role=webserver** label treats both of these pods as part of the same group.

The [Kubernetes documentation](#) has more information on labels.

3.3.4. Endpoints

The servers that back a service are called its endpoints, and are specified by an object of type **Endpoints** with the same name as the service. When a service is backed by pods, those pods are normally specified by a label selector in the service specification, and OpenShift Dedicated automatically creates the Endpoints object pointing to those pods.

In some cases, you may want to create a service but have it be backed by external hosts rather than by pods in the OpenShift Dedicated cluster. In this case, you can leave out the **selector** field in the service, and [create the Endpoints object manually](#).

Note that OpenShift Dedicated will not let most users manually create an Endpoints object that points to an IP address in the network blocks reserved for pod and service IPs. Only cluster admins or other users with [permission to create resources under endpoints/restricted](#) can create such Endpoint objects.

3.4. PROJECTS AND USERS

3.4.1. Users

Interaction with OpenShift Dedicated is associated with a user. An OpenShift Dedicated user object represents an actor which may be granted permissions in the system by adding roles to them or to their groups.

Several types of users can exist:

Regular users	This is the way most interactive OpenShift Dedicated users will be represented. Regular users are created automatically in the system upon first login, or can be created via the API. Regular users are represented with the User object. Examples: joe alice
System users	Many of these are created automatically when the infrastructure is defined, mainly for the purpose of enabling the infrastructure to interact with the API securely. They include a cluster administrator (with access to everything), a per-node user, users for use by routers and registries, and various others. Finally, there is an anonymous system user that is used by default for unauthenticated requests. Examples: system:admin system:openshift-registry system:node:node1.example.com
Service accounts	These are special system users associated with projects; some are created automatically when the project is first created, while project administrators can create more for the purpose of defining access to the contents of each project . Service accounts are represented with the ServiceAccount object. Examples: system:serviceaccount:default:deployer system:serviceaccount:foo:builder

Every user must [authenticate](#) in some way in order to access OpenShift Dedicated. API requests with no authentication or invalid authentication are authenticated as requests by the **anonymous** system user. Once authenticated, policy determines what the user is [authorized](#) to do.

3.4.2. Namespaces

A Kubernetes namespace provides a mechanism to scope resources in a cluster. In OpenShift Dedicated, a [project](#) is a Kubernetes namespace with additional annotations.

Namespaces provide a unique scope for:

- Named resources to avoid basic naming collisions.
- Delegated management authority to trusted users.
- The ability to limit community resource consumption.

Most objects in the system are scoped by namespace, but some are excepted and have no namespace, including nodes and users.

The [Kubernetes documentation](#) has more information on namespaces.

3.4.3. Projects

A project is a Kubernetes namespace with additional annotations, and is the central vehicle by which access to resources for regular users is managed. A project allows a community of users to organize and manage their content in isolation from other communities. Users must be given access to projects by administrators, or if allowed to create projects, automatically have access to their own projects.

Projects can have a separate **name**, **displayName**, and **description**.

- The mandatory **name** is a unique identifier for the project and is most visible when using the CLI tools or API. The maximum name length is 63 characters.
- The optional **displayName** is how the project is displayed in the web console (defaults to **name**).
- The optional **description** can be a more detailed description of the project and is also visible in the web console.

Each project scopes its own set of:

Objects	Pods, services, replication controllers, etc.
Policies	Rules for which users can or cannot perform actions on objects.
Constraints	Quotas for each kind of object that can be limited.
Service accounts	Service accounts act automatically with designated access to objects in the project.

Cluster administrators can [create projects](#) and delegate administrative rights for the project to any member of the user community. Cluster administrators can also allow developers to create their own projects.

Developers and administrators can [interact with projects](#) using [the CLI](#) or the [web console](#).

3.4.3.1. Projects provided at installation

OpenShift Dedicated comes with a number of projects out of the box, and **openshift** is the most essential to users:

openshift A user-facing project, mainly for housing objects for day-to-day tasks. These include any application objects for access by multiple projects, such as templates and images. These objects should be those that do not require communication between the pods.

3.5. BUILDS AND IMAGE STREAMS

3.5.1. Builds

A *build* is the process of transforming input parameters into a resulting object. Most often, the process is used to transform input parameters or source code into a runnable image. A **BuildConfig** object is the definition of the entire build process.

OpenShift Dedicated leverages Kubernetes by creating Docker-formatted containers from build images and pushing them to a [container registry](#).

Build objects share common characteristics: inputs for a build, the need to complete a build process, logging the build process, publishing resources from successful builds, and publishing the final status of the build. Builds take advantage of resource restrictions, specifying limitations on resources such as CPU usage, memory usage, and build or pod execution time.

The OpenShift Dedicated build system provides extensible support for *build strategies* that are based on selectable types specified in the build API. There are three primary build strategies available:

- [Docker build](#)
- [Source-to-Image \(S2I\) build](#)
- [Custom build](#)

By default, Docker builds and S2I builds are supported.

The resulting object of a build depends on the builder used to create it. For Docker and S2I builds, the resulting objects are runnable images. For Custom builds, the resulting objects are whatever the builder image author has specified.

Additionally, the [Pipeline build](#) strategy can be used to implement sophisticated workflows:

- continuous integration
- continuous deployment

For a list of build commands, see the [Developer's Guide](#).

For more information on how OpenShift Dedicated leverages Docker for builds, see the [upstream documentation](#).

3.5.1.1. Docker Build

The Docker build strategy invokes the `docker build` command, and it therefore expects a repository with a **Dockerfile** and all required artifacts in it to produce a runnable image.

3.5.1.2. Source-to-Image (S2I) Build

Source-to-Image (S2I) is a tool for building reproducible, Docker-formatted container images. It produces ready-to-run images by injecting application source into a container image and assembling a new image. The new image incorporates the base image (the builder) and built source and is ready to use with the `docker run` command. S2I supports incremental builds, which re-use previously downloaded dependencies, previously built artifacts, etc.

The advantages of S2I include the following:

Image flexibility	S2I scripts can be written to inject application code into almost any existing Docker-formatted container image, taking advantage of the existing ecosystem. Note that, currently, S2I relies on tar to inject application source, so the image needs to be able to process tarred content.
Speed	With S2I, the assemble process can perform a large number of complex operations without creating a new layer at each step, resulting in a fast process. In addition, S2I scripts can be written to re-use artifacts stored in a previous version of the application image, rather than having to download or build them each time the build is run.
Patchability	S2I allows you to rebuild the application consistently if an underlying image needs a patch due to a security issue.
Operational efficiency	By restricting build operations instead of allowing arbitrary actions, as a Dockerfile would allow, the PaaS operator can avoid accidental or intentional abuses of the build system.
Operational security	Building an arbitrary Dockerfile exposes the host system to root privilege escalation. This can be exploited by a malicious user because the entire Docker build process is run as a user with Docker privileges. S2I restricts the operations performed as a root user and can run the scripts as a non-root user.
User efficiency	S2I prevents developers from performing arbitrary yum install type operations, which could slow down development iteration, during their application build.
Ecosystem	S2I encourages a shared ecosystem of images where you can leverage best practices for your applications.
Reproducibility	Produced images can include all inputs including specific versions of build tools and dependencies. This ensures that the image can be reproduced precisely.

3.5.1.3. Custom Build

The Custom build strategy allows developers to define a specific builder image responsible for the entire build process. Using your own builder image allows you to customize your build process.

A **Custom builder image** is a plain Docker-formatted container image embedded with build process logic, for example for building RPMs or base images. The **openshift/origin-custom-docker-builder** image is available on the [Docker Hub](#) registry as an example implementation of a Custom builder image.

3.5.1.4. Pipeline Build

The Pipeline build strategy allows developers to define a *Jenkins pipeline* for execution by the Jenkins pipeline plugin. The build can be started, monitored, and managed by OpenShift Dedicated in the same way as any other build type.

Pipeline workflows are defined in a Jenkinsfile, either embedded directly in the build configuration, or supplied in a Git repository and referenced by the build configuration.

The first time a project defines a build configuration using a Pipeline strategy, OpenShift Dedicated instantiates a Jenkins server to execute the pipeline. Subsequent Pipeline build configurations in the project share this Jenkins server.



NOTE

The Jenkins server is not automatically removed, even if all Pipeline build configurations are deleted. It must be manually deleted by the user.

For more information about Jenkins Pipelines, see the [Jenkins documentation](#).

3.5.2. Image Streams

An image stream and its associated tags provide an abstraction for referencing [Docker images](#) from within OpenShift Dedicated. The image stream and its tags allow you to see what images are available and ensure that you are using the specific image you need even if the image in the repository changes.

Image streams do not contain actual image data, but present a single virtual view of related images, similar to an image repository.

You can configure [Builds](#) and [Deployments](#) to watch an image stream for notifications when new images are added and react by performing a Build or Deployment, respectively.

For example, if a Deployment is using a certain image and a new version of that image is created, a Deployment could be automatically performed to pick up the new version of the image.

However, if the image stream tag used by the Deployment or Build is not updated, then even if the Docker image in the Docker registry is updated, the Build or Deployment will continue using the previous (presumably known good) image.

The source images can be stored in any of the following:

- OpenShift Dedicated's [integrated registry](#)
- An external registry, for example `registry.access.redhat.com` or `hub.docker.com`
- Other image streams in the OpenShift Dedicated cluster

When you define an object that references an image stream tag (such as a Build or Deployment configuration), you point to an image stream tag, not the Docker repository. When you Build or Deploy your application, OpenShift Dedicated queries the Docker repository using the image stream tag to locate the associated ID of the image and uses that exact image.

The image stream metadata is stored in the etcd instance along with other cluster information.

The following image stream contains two tags: **34** which points to a Python v3.4 image and **35** which points to a Python v3.5 image:

```

oc describe is python
Name:      python
Namespace: imagestream
Created:   25 hours ago
Labels:    app=python
Annotations: openshift.io/generated-by=OpenShiftWebConsole
            openshift.io/image.dockerRepositoryCheck=2017-10-03T19:48:00Z
Docker Pull Spec: docker-registry.default.svc:5000/imagestream/python
Image Lookup: local=false
Unique Images: 2
Tags:        2

34
  tagged from centos/python-34-centos7

  * centos/python-34-
centos7@sha256:28178e2352d31f240de1af1370be855db33ae9782de737bb005247d8791
a54d0
    14 seconds ago

35
  tagged from centos/python-35-centos7

  * centos/python-35-
centos7@sha256:2efb79ca3ac9c9145a63675fb0c09220ab3b8d4005d35e0644417ee5525
48b10
    7 seconds ago

```

Using image streams has several significant benefits:

- You can tag, rollback a tag, and quickly deal with images, without having to re-push using the command line.
- You can trigger Builds and Deployments when a new image is pushed to the registry. Also, OpenShift Dedicated has generic triggers for other resources (such as Kubernetes objects).
- You can [mark a tag for periodic re-import](#). If the source image has changed, that change is picked up and reflected in the image stream, which triggers the Build and/or Deployment flow, depending upon the Build or Deployment configuration.
- You can share images using fine-grained access control and quickly distribute images across your teams.
- If the source image changes, the image stream tag will still point to a known-good version of the image, ensuring that your application will not break unexpectedly.
- You can configure security around who can view and use the images through permissions on the image stream objects.
- Users that lack permission to read or list images on the cluster level can still retrieve the images tagged in a project using image streams.

For a curated set of image streams, see the [OpenShift Image Streams and Templates library](#).

When using image streams, it is important to understand what the image stream tag is pointing to and how changes to tags and images can affect you. For example:

- If your image stream tag points to a Docker image tag, you need to understand how that Docker image tag is updated. For example, a Docker image tag **docker.io/ruby:2.4** will probably always point to a v2.4 ruby image. But, a Docker image tag **docker.io/ruby:latest** will probably change with major versions. So, the Docker image tag that a image stream tag points to can tell you how stable the image stream tag will be, if you choose to reference it.
- If your image stream tag follows another image stream tag (it does not point directly to a docker image tag), it is possible that the image stream tag will be updated to follow a different image stream tag in the future. Again, this could result in picking up an incompatible version change.

3.5.2.1. Important terms

Docker repository

A collection of related docker images and tags identifying them. For example, the OpenShift Jenkins images are in a Docker repository:

```
docker.io/openshift/jenkins-2-centos7
```

Docker registry

A content server that can store and service images from Docker repositories. For example:

```
registry.access.redhat.com
```

Docker image

A specific set of content that can be run as a container. Usually associated with a particular tag within a Docker repository.

Docker image tag

A label applied to a Docker image in a repository that distinguishes a specific image. For example, here **3.6.0** is a tag:

```
docker.io/openshift/jenkins-2-centos7:3.6.0
```



NOTE

A Docker image tag can be updated to point to new Docker image content at any time.

Docker image ID

A SHA (Secure Hash Algorithm) code that can be used to pull an image. For example:

```
docker.io/openshift/jenkins-2-centos7@sha256:ab312bda324
```



NOTE

A SHA image ID cannot change. A specific SHA identifier always references the exact same docker image content.

Image stream

An OpenShift Dedicated object that contains pointers to any number of Docker-formatted container images identified by tags. You can think of an image stream as equivalent to a Docker repository.

Image stream tag

A named pointer to an image in an image stream. An image stream tag is similar to a Docker image tag. See [Image Stream Tag](#) below.

Image stream image

An image that allows you to retrieve a specific Docker image from a particular image stream where it is tagged. An image stream image is an API resource object that pulls together some metadata about a particular image SHA identifier. See [Image Stream Images](#) below.

Image stream trigger

A trigger that causes a specific action when an image stream tag changes. For example, importing can cause the value of the tag to change, which causes a trigger to fire when there are Deployments, Builds, or other resources listening for those. See [Image Stream Triggers](#) below.

3.5.2.2. Configuring Image Streams

An image stream object file contains the following elements.



NOTE

See the [Developer Guide](#) for details on managing images and image streams.

Image Stream Object Definition

```
apiVersion: v1
kind: ImageStream
metadata:
  annotations:
    openshift.io/generated-by: OpenShiftNewApp
  creationTimestamp: 2017-09-29T13:33:49Z
  generation: 1
  labels:
    app: ruby-sample-build
    template: application-template-stibuild
  name: origin-ruby-sample 1
  namespace: test
  resourceVersion: "633"
  selflink: /oapi/v1/namespaces/test/imagestreams/origin-ruby-sample
  uid: ee2b9405-c68c-11e5-8a99-525400f25e34
spec: {}
status:
  dockerImageRepository: 172.30.56.218:5000/test/origin-ruby-sample 2
  tags:
    - items:
        - created: 2017-09-02T10:15:09Z
          dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7d
d13d 3
          generation: 2
          image:
sha256:909de62d1f609a717ec433cc25ca5cf00941545c83a01fb31527771e1fab3fc5 4
        - created: 2017-09-29T13:40:11Z
          dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:909de62d1f609a717ec433cc25ca5cf00941545c83a01fb31527771e1fab
3fc5
```

```

    generation: 1
    image:
sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d
    tag: latest 5

```

- 1 The name of the image stream.
- 2 Docker repository path where new images can be pushed to add/update them in this image stream.
- 3 The SHA identifier that this image stream tag currently references. Resources that reference this image stream tag use this identifier.
- 4 The SHA identifier that this image stream tag previously referenced. Can be used to rollback to an older image.
- 5 The image stream tag name.

For a sample build configuration that references an image stream, see [What Is a BuildConfig?](#) in the **Strategy** stanza of the configuration.

For a sample deployment configuration that references an image stream, see [Creating a Deployment Configuration](#) in the **Strategy** stanza of the configuration.

3.5.2.3. Image Stream Images

An *image stream image* points from within an image stream to a particular image ID.

Image stream images allow you to retrieve metadata about an image from a particular image stream where it is tagged.

Image stream image objects are automatically created in OpenShift Dedicated whenever you import or tag an image into the image stream. You should never have to explicitly define an image stream image object in any image stream definition that you use to create image streams.

The image stream image consists of the image stream name and image ID from the repository, delimited by an @ sign:

```
<image-stream-name>@<image-id>
```

To refer to the image in the [image stream object example above](#), the image stream image looks like:

```
origin-ruby-
sample@sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7d
d13d
```

3.5.2.4. Image Stream Tags

An *image stream tag* is a named pointer to an image in an *image stream*. It is often abbreviated as *istag*. An image stream tag is used to reference or retrieve an image for a given image stream and tag.

Image stream tags can reference any local or externally managed image. It contains a history of images represented as a stack of all images the tag ever pointed to. Whenever a new or existing image is tagged under particular image stream tag, it is placed at the first position in the history stack. The image

previously occupying the top position will be available at the second position, and so forth. This allows for easy rollbacks to make tags point to historical images again.

The following image stream tag is from the the [image stream object example above](#):

Image Stream Tag with Two Images in its History

```
tags:
- items:
  - created: 2017-09-02T10:15:09Z
    dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7d
d13d
    generation: 2
    image:
sha256:909de62d1f609a717ec433cc25ca5cf00941545c83a01fb31527771e1fab3fc5
  - created: 2017-09-29T13:40:11Z
    dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:909de62d1f609a717ec433cc25ca5cf00941545c83a01fb31527771e1fab
3fc5
    generation: 1
    image:
sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d
    tag: latest
```

Image stream tags can be *permanent* tags or *tracking* tags.

- *Permanent tags* are version-specific tags that point to a particular version of an image, such as Python 3.5.
- *Tracking tags* are reference tags that follow another image stream tag and could be updated in the future to change which image they follow, much like a symlink. Note that these new levels are not guaranteed to be backwards-compatible.
For example, the **latest** image stream tags that ship with OpenShift Dedicated are tracking tags. This means consumers of the **latest** image stream tag will be updated to the newest level of the framework provided by the image when a new level becomes available. A **latest** image stream tag to **v3.6** could be changed to **v3.7** at any time. It is important to be aware that these **latest** image stream tags behave differently than the Docker **latest** tag. The **latest** image stream tag, in this case, does not point to the latest image in the Docker repository. It points to another image stream tag, which might not be the latest version of an image. For example, if the **latest** image stream tag points to **v3.2** of an image, when the **3.3** version is released, the **latest** tag is not automatically updated to **v3.3**, and remains at **v3.2** until it is manually updated to point to a **v3.3** image stream tag.



NOTE

Tracking tags are limited to a single image stream and cannot reference other image streams.

You can create your own image stream tags for your own needs. See the [Recommended Tagging Conventions](#).

The image stream tag is composed of the name of the image stream and a tag, separated by a colon:

```
<image stream name>:<tag>
```

For example, to refer to the

sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d image in the [image stream object example above](#), the image stream tag would be:

```
origin-ruby-sample:latest
```

3.5.2.5. Image Stream Change Triggers

Image stream triggers allow your Builds and Deployments to be automatically invoked when a new version of an upstream image is available.

For example, Builds and Deployments can be automatically started when an image stream tag is modified. This is achieved by monitoring that particular image stream tag and notifying the Build or Deployment when a change is detected.

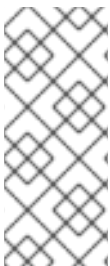
The **ImageChange** trigger results in a new replication controller whenever the content of an [image stream tag](#) changes (when a new version of the image is pushed).

Example 3.3. An ImageChange Trigger

```
triggers:
  - type: "ImageChange"
    imageChangeParams:
      automatic: true 1
      from:
        kind: "ImageStreamTag"
        name: "origin-ruby-sample:latest"
        namespace: "myproject"
      containerNames:
        - "helloworld"
```

1 If the `imageChangeParams.automatic` field is set to **false**, the trigger is disabled.

With the above example, when the **latest** tag value of the **origin-ruby-sample** image stream changes and the new image value differs from the current image specified in the deployment configuration's **helloworld** container, a new replication controller is created using the new image for the **helloworld** container.

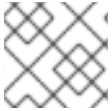


NOTE

If an **ImageChange** trigger is defined on a deployment configuration (with a **ConfigChange** trigger and **automatic=false**, or with **automatic=true**) and the **ImageStreamTag** pointed by the **ImageChange** trigger does not exist yet, then the initial deployment process will automatically start as soon as an image is imported or pushed by a build to the **ImageStreamTag**.

3.5.2.6. Image Stream Mappings

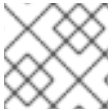
When the [integrated registry](#) receives a new image, it creates and sends an image stream mapping to OpenShift Dedicated, providing the image's project, name, tag, and image metadata.



NOTE

Configuring image stream mappings is an advanced feature.

This information is used to create a new image (if it does not already exist) and to tag the image into the image stream. OpenShift Dedicated stores complete metadata about each image, such as commands, entry point, and environment variables. Images in OpenShift Dedicated are immutable and the maximum name length is 63 characters.



NOTE

See the [Developer Guide](#) for details on manually tagging images.

The following image stream mapping example results in an image being tagged as **test/origin-ruby-sample:latest**:

Image Stream Mapping Object Definition

```
apiVersion: v1
kind: ImageStreamMapping
metadata:
  creationTimestamp: null
  name: origin-ruby-sample
  namespace: test
tag: latest
image:
  dockerImageLayers:
    - name:
sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef
  size: 0
    - name:
sha256:ee1dd2cb6df21971f4af6de0f1d7782b81fb63156801cfde2bb47b4247c23c29
  size: 196634330
    - name:
sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef
  size: 0
    - name:
sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef
  size: 0
    - name:
sha256:ca062656bff07f18bff46be00f40cfbb069687ec124ac0aa038fd676cfaea092
  size: 177723024
    - name:
sha256:63d529c59c92843c395befd065de516ee9ed4995549f8218eac6ff088bfa6b6e
  size: 55679776
    - name:
sha256:92114219a04977b5563d7dff71ec4caa3a37a15b266ce42ee8f43dba9798c966
  size: 11939149
  dockerImageMetadata:
    Architecture: amd64
    Config:
```

```

Cmd:
- /usr/libexec/s2i/run
Entrypoint:
- container-entrpoint
Env:
- RACK_ENV=production
- OPENSIFT_BUILD_NAMESPACE=test
- OPENSIFT_BUILD_SOURCE=https://github.com/openshift/ruby-hello-
world.git
- EXAMPLE=sample-app
- OPENSIFT_BUILD_NAME=ruby-sample-build-1
- PATH=/opt/app-root/src/bin:/opt/app-
root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
- STI_SCRIPTS_URL=image:///usr/libexec/s2i
- STI_SCRIPTS_PATH=/usr/libexec/s2i
- HOME=/opt/app-root/src
- BASH_ENV=/opt/app-root/etc/scl_enable
- ENV=/opt/app-root/etc/scl_enable
- PROMPT_COMMAND=. /opt/app-root/etc/scl_enable
- RUBY_VERSION=2.2
ExposedPorts:
  8080/tcp: {}
Labels:
  build-date: 2015-12-23
  io.k8s.description: Platform for building and running Ruby 2.2
applications
  io.k8s.display-name: 172.30.56.218:5000/test/origin-ruby-
sample:latest
  io.openshift.build.commit.author: Ben Parees
<bparees@users.noreply.github.com>
  io.openshift.build.commit.date: Wed Jan 20 10:14:27 2016 -0500
  io.openshift.build.commit.id:
00cad392d39d5ef9117cbc8a31db0889eedd442
  io.openshift.build.commit.message: 'Merge pull request #51 from
php-coder/fix_url_and_sti'
  io.openshift.build.commit.ref: master
  io.openshift.build.image: centos/ruby-22-
centos7@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e898
6b28e
  io.openshift.build.source-location:
https://github.com/openshift/ruby-hello-world.git
  io.openshift.builder-base-version: 8d95148
  io.openshift.builder-version:
8847438ba06307f86ac877465eadc835201241df
  io.openshift.s2i.scripts-url: image:///usr/libexec/s2i
  io.openshift.tags: builder,ruby,ruby22
  io.s2i.scripts-url: image:///usr/libexec/s2i
  license: GPLv2
  name: CentOS Base Image
  vendor: CentOS
User: "1001"
WorkingDir: /opt/app-root/src
Container:
86e9a4a3c760271671ab913616c51c9f3cea846ca524bf07c04a6f6c9e103a76
ContainerConfig:
  AttachStdout: true

```

```

Cmd:
- /bin/sh
- -c
- tar -C /tmp -xf - && /usr/libexec/s2i/assemble
Entrypoint:
- container-entrpoint
Env:
- RACK_ENV=production
- OPENSIFT_BUILD_NAME=ruby-sample-build-1
- OPENSIFT_BUILD_NAMESPACE=test
- OPENSIFT_BUILD_SOURCE=https://github.com/openshift/ruby-hello-
world.git
- EXAMPLE=sample-app
- PATH=/opt/app-root/src/bin:/opt/app-
root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
- STI_SCRIPTS_URL=image:///usr/libexec/s2i
- STI_SCRIPTS_PATH=/usr/libexec/s2i
- HOME=/opt/app-root/src
- BASH_ENV=/opt/app-root/etc/scl_enable
- ENV=/opt/app-root/etc/scl_enable
- PROMPT_COMMAND=. /opt/app-root/etc/scl_enable
- RUBY_VERSION=2.2
ExposedPorts:
  8080/tcp: {}
Hostname: ruby-sample-build-1-build
Image: centos/ruby-22-
centos7@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e898
6b28e
  OpenStdin: true
  StdinOnce: true
  User: "1001"
  WorkingDir: /opt/app-root/src
Created: 2016-01-29T13:40:00Z
DockerVersion: 1.8.2.fc21
Id: 9d7fd5e2d15495802028c569d544329f4286dcd1c9c085ff5699218dbaa69b43
Parent:
57b08d979c86f4500dc8cad639c9518744c8dd39447c055a3517dc9c18d6fccd
Size: 441976279
apiVersion: "1.0"
kind: DockerImage
dockerImageMetadataVersion: "1.0"
dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7d
d13d

```

3.5.2.7. Working with Image Streams

The following sections describe how to use image streams and image stream tags. For more information on working with image streams, see [Managing Images](#).

3.5.2.7.1. Getting Information about Image Streams

To get general information about the image stream and detailed information about all the tags it is pointing to, use the following command:

```
oc describe is/<image-name>
```

For example:

```
oc describe is/python

Name:      python
Namespace: default
Created:   About a minute ago
Labels:    <none>
Annotations: openshift.io/image.dockerRepositoryCheck=2017-10-02T17:05:11Z
Docker Pull Spec: docker-registry.default.svc:5000/default/python
Image Lookup: local=false
Unique Images: 1
Tags:      1

3.5
  tagged from centos/python-35-centos7

  * centos/python-35-
centos7@sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25
    About a minute ago
```

To get all the information available about particular image stream tag:

```
oc describe istag/<image-stream>:<tag-name>
```

For example:

```
oc describe istag/python:latest

Image Name: sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25
Docker Image: centos/python-35-
centos7@sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25
Name:      sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25
Created:   2 minutes ago
Image Size: 251.2 MB (first layer 2.898 MB, last binary layer 72.26 MB)
Image Created: 2 weeks ago
Author:    <none>
Arch:      amd64
Entrypoint: container-entrypoint
Command:   /bin/sh -c $STI_SCRIPTS_PATH/usage
Working Dir: /opt/app-root/src
User:      1001
Exposes Ports: 8080/tcp
Docker Labels: build-date=20170801
```

**NOTE**

More information is output than shown.

3.5.2.7.2. Adding Additional Tags to an Image Stream

To add a tag that points to one of the existing tags, you can use the **oc tag** command:

```
oc tag <image-name:tag> <image-name:tag>
```

For example:

```
oc tag python:3.5 python:latest
```

```
Tag python:latest set to
python@sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25.
```

Use the **oc describe** command to confirm the the image stream has two tags, one (**3.5**) pointing at the external Docker image and another tag (**latest**) pointing to the same image because it was created based on the first tag.

```
oc describe is/python
```

```
Name:      python
Namespace: default
Created:   5 minutes ago
Labels:    <none>
Annotations: openshift.io/image.dockerRepositoryCheck=2017-10-02T17:05:11Z
Docker Pull Spec: docker-registry.default.svc:5000/default/python
Image Lookup: local=false
Unique Images: 1
Tags:        2

latest
  tagged from
  python@sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25

  * centos/python-35-
  centos7@sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25
    About a minute ago

3.5
  tagged from centos/python-35-centos7

  * centos/python-35-
  centos7@sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25
    5 minutes ago
```

3.5.2.7.3. Adding Tags for an External Image

Use the **oc tag** command for all tag-related operations, such as adding tags pointing to internal or external images:

```
oc tag <repository/image> <image-name:tag>
```

For example, this command maps the **docker.io/python:3.6.0** image to the **3.6** tag in the **python** image stream.

```
oc tag docker.io/python:3.6.0 python:3.6
```

```
Tag python:3.6 set to docker.io/python:3.6.0.
```

If the external image is secured, you will need to create a secret with credentials for accessing that registry. See [Importing Images from Private Registries](#) for more details.

3.5.2.7.4. Updating an Image Stream Tag

To update a tag to reflect another tag in an image stream:

```
oc tag <image-name:tag> <image-name:latest>
```

For example, the following updates the **latest** tag to reflect the **3.6** tag in an image stream:

```
oc tag python:3.6 python:latest
Tag python:latest set to
python@sha256:438208801c4806548460b27bd1fbc7bb188273d13871ab43f.
```

3.5.2.7.5. Removing Image Stream Tags from an Image Stream

To remove old tags from an image stream:

```
oc tag -d <image-name:tag>
```

For example:

```
oc tag -d python:3.5
Deleted tag default/python:3.5.
```

3.5.2.7.6. Configuring Periodic Importing of Tags

When working with an external Docker registry, to periodically re-import an image (such as, to get latest security updates), use the **--scheduled** flag:

```
oc tag <repository/image> <image-name:tag> --scheduled
```

For example:

```
oc tag docker.io/python:3.6.0 python:3.6 --scheduled
Tag python:3.6 set to import docker.io/python:3.6.0 periodically.
```

This command causes OpenShift Dedicated to periodically update this particular image stream tag. This period is a cluster-wide setting set to 15 minutes by default.

To remove the periodic check, re-run above command but omit the **--scheduled** flag. This will reset its behavior to default.

```
oc tag <repository/image> <image-name:tag>
```

3.6. DEPLOYMENTS

3.6.1. Replication controllers

A [replication controller](#) ensures that a specified number of replicas of a pod are running at all times. If pods exit or are deleted, the replication controller acts to instantiate more up to the defined number. Likewise, if there are more running than desired, it deletes as many as necessary to match the defined amount.

A replication controller configuration consists of:

1. The number of replicas desired (which can be adjusted at runtime).
2. A pod definition to use when creating a replicated pod.
3. A selector for identifying managed pods.

A selector is a set of labels assigned to the pods that are managed by the replication controller. These labels are included in the pod definition that the replication controller instantiates. The replication controller uses the selector to determine how many instances of the pod are already running in order to adjust as needed.

The replication controller does not perform auto-scaling based on load or traffic, as it does not track either. Rather, this would require its replica count to be adjusted by an external auto-scaler.

A replication controller is a core Kubernetes object called **ReplicationController**.

The following is an example **ReplicationController** definition:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend-1
spec:
  replicas: 1
  selector:
    name: frontend
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
      - image: openshift/hello-openshift
        name: helloworld
        ports:
        - containerPort: 8080
          protocol: TCP
      restartPolicy: Always
```

- 1 The number of copies of the pod to run.
- 2 The label selector of the pod to run.
- 3 A template for the pod the controller creates.
- 4 Labels on the pod should include those from the label selector.

- 5 The maximum name length after expanding any parameters is 63 characters.

3.6.2. Replica set

Similar to a [replication controller](#), a replica set ensures that a specified number of pod replicas are running at any given time. The difference between a replica set and a replication controller is that a replica set supports set-based selector requirements whereas a replication controller only supports equality-based selector requirements.



NOTE

Only use replica sets if you require custom update orchestration or do not require updates at all, otherwise, use [Deployments](#). Replica sets can be used independently, but are used by deployments to orchestrate pod creation, deletion, and updates. Deployments manage their replica sets automatically, provide declarative updates to pods, and do not have to manually manage the replica sets that they create.

A replica set is a core Kubernetes object called **ReplicaSet**.

The following is an example **ReplicaSet** definition:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend-1
  labels:
    tier: frontend
spec:
  replicas: 3
  selector: 1
    matchLabels: 2
      tier: frontend
    matchExpressions: 3
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
        - image: openshift/hello-openshift
          name: helloworld
          ports:
            - containerPort: 8080
              protocol: TCP
          restartPolicy: Always
```

- 1 A label query over a set of resources. The result of **matchLabels** and **matchExpressions** are logically conjoined.

- 2 Equality-based selector to specify resources with labels that match the selector.

- 3

Set-based selector to filter keys. This selects all resources with key equal to **tier** and value equal to **frontend**.

3.6.3. Jobs

A job is similar to a replication controller, in that its purpose is to create pods for specified reasons. The difference is that replication controllers are designed for pods that will be continuously running, whereas jobs are for one-time pods. A job tracks any successful completions and when the specified amount of completions have been reached, the job itself is completed.

The following example computes π to 2000 places, prints it out, then completes:

```
apiVersion: extensions/v1
kind: Job
metadata:
  name: pi
spec:
  selector:
    matchLabels:
      app: pi
  template:
    metadata:
      name: pi
      labels:
        app: pi
    spec:
      containers:
        - name: pi
          image: perl
          command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
          restartPolicy: Never
```

See the [Jobs](#) topic for more information on how to use jobs.

3.6.4. Deployments and Deployment Configurations

Building on replication controllers, OpenShift Dedicated adds expanded support for the software development and deployment lifecycle with the concept of deployments. In the simplest case, a deployment just creates a new replication controller and lets it start up pods. However, OpenShift Dedicated deployments also provide the ability to transition from an existing deployment of an image to a new one and also define hooks to be run before or after creating the replication controller.

The OpenShift Dedicated **DeploymentConfig** object defines the following details of a deployment:

1. The elements of a **ReplicationController** definition.
2. Triggers for creating a new deployment automatically.
3. The strategy for transitioning between deployments.
4. Life cycle hooks.

Each time a deployment is triggered, whether manually or automatically, a deployer pod manages the deployment (including scaling down the old replication controller, scaling up the new one, and running

hooks). The deployment pod remains for an indefinite amount of time after it completes the deployment in order to retain its logs of the deployment. When a deployment is superseded by another, the previous replication controller is retained to enable easy rollback if needed.

For detailed instructions on how to create and interact with deployments, refer to [Deployments](#).

Here is an example **DeploymentConfig** definition with some omissions and callouts:

```
apiVersion: v1
kind: DeploymentConfig
metadata:
  name: frontend
spec:
  replicas: 5
  selector:
    name: frontend
  template: { ... }
  triggers:
    - type: ConfigChange 1
    - imageChangeParams:
        automatic: true
        containerNames:
          - helloworld
        from:
          kind: ImageStreamTag
          name: hello-openshift:latest
        type: ImageChange 2
  strategy:
    type: Rolling 3
```

- 1 A **ConfigChange** trigger causes a new deployment to be created any time the replication controller template changes.
- 2 An **ImageChange** trigger causes a new deployment to be created each time a new version of the backing image is available in the named image stream.
- 3 The default **Rolling** strategy makes a downtime-free transition between deployments.

3.7. TEMPLATES

3.7.1. Overview

A template describes a set of [objects](#) that can be parameterized and processed to produce a list of objects for creation by OpenShift Dedicated. The objects to create can include anything that users have permission to create within a project, for example [services](#), [build configurations](#), and [deployment configurations](#). A template may also define a set of [labels](#) to apply to every object defined in the template.

See the [template guide](#) for details about creating and using templates.

CHAPTER 4. ADDITIONAL CONCEPTS

4.1. AUTHENTICATION

4.1.1. Overview

The authentication layer identifies the user associated with requests to the OpenShift Dedicated API. The authorization layer then uses information about the requesting user to determine if the request should be allowed.

4.1.2. Users and Groups

A *user* in OpenShift Dedicated is an entity that can make requests to the OpenShift Dedicated API. Typically, this represents the account of a developer or administrator that is interacting with OpenShift Dedicated.

A user can be assigned to one or more *groups*, each of which represent a certain set of users. Groups are useful when managing authorization policies to grant permissions to multiple users at once, for example allowing access to [objects](#) within a [project](#), versus granting them to users individually.

In addition to explicitly defined groups, there are also system groups, or *virtual groups*, that are automatically provisioned by OpenShift.

In the default set of virtual groups, note the following in particular:

Virtual Group	Description
system:authenticated	Automatically associated with all authenticated users.
system:authenticated:oauth	Automatically associated with all users authenticated with an OAuth access token.
system:unauthenticated	Automatically associated with all unauthenticated users.

4.1.3. API Authentication

Requests to the OpenShift Dedicated API are authenticated using the following methods:

OAuth Access Tokens

- Obtained from the OpenShift Dedicated OAuth server using the `<master>/oauth/authorize` and `<master>/oauth/token` endpoints.
- Sent as an **Authorization: Bearer...** header
- Sent as an **access_token=...** query parameter for websocket requests prior to OpenShift Dedicated server version 3.6.
- Sent as a websocket subprotocol header in the form **base64url.bearer.authorization.k8s.io.<base64url-encoded-token>** for websocket requests in OpenShift Dedicated server version 3.6 and later.

X.509 Client Certificates

- Requires a HTTPS connection to the API server.
- Verified by the API server against a trusted certificate authority bundle.
- The API server creates and distributes certificates to controllers to authenticate themselves.

Any request with an invalid access token or an invalid certificate is rejected by the authentication layer with a 401 error.

If no access token or certificate is presented, the authentication layer assigns the **system:anonymous** virtual user and the **system:unauthenticated** virtual group to the request. This allows the authorization layer to determine which requests, if any, an anonymous user is allowed to make.

4.1.3.1. Impersonation

A request to the OpenShift Dedicated API can include an **Impersonate-User** header, which indicates that the requester wants to have the request handled as though it came from the specified user. You impersonate a user by adding the `--as=<user>` flag to requests.

Before User A can impersonate User B, User A is authenticated. Then, an authorization check occurs to ensure that User A is allowed to impersonate the user named User B. If User A is requesting to impersonate a service account, **system:serviceaccount:namespace:name**, OpenShift Dedicated confirms that User A can impersonate the **serviceaccount** named **name** in **namespace**. If the check fails, the request fails with a 403 (Forbidden) error code.

By default, project administrators and editors can impersonate service accounts in their namespace.

4.1.4. OAuth

The OpenShift Dedicated master includes a built-in OAuth server. Users obtain OAuth access tokens to authenticate themselves to the API.

When a person requests a new OAuth token, the OAuth server uses the configured identity provider to determine the identity of the person making the request.

It then determines what user that identity maps to, creates an access token for that user, and returns the token for use.

4.1.4.1. OAuth Clients

Every request for an OAuth token must specify the OAuth client that will receive and use the token. The following OAuth clients are automatically created when starting the OpenShift Dedicated API:

OAuth Client	Usage
openshift-web-console	Requests tokens for the web console.
openshift-browser-client	Requests tokens at <master>/oauth/token/request with a user-agent that can handle interactive logins.

OAuth Client	Usage
openshift-challenging-client	Requests tokens with a user-agent that can handle WWW-Authenticate challenges.

To register additional clients:

```
$ oc create -f <(echo '
kind: OAuthClient
apiVersion: v1
metadata:
  name: demo ❶
secret: "... " ❷
redirectURIs:
  - "http://www.example.com/" ❸
grantMethod: prompt ❹
')
```

- ❶ The **name** of the OAuth client is used as the **client_id** parameter when making requests to **<master>/oauth/authorize** and **<master>/oauth/token**.
- ❷ The **secret** is used as the **client_secret** parameter when making requests to **<master>/oauth/token**.
- ❸ The **redirect_uri** parameter specified in requests to **<master>/oauth/authorize** and **<master>/oauth/token** must be equal to (or prefixed by) one of the URLs in **redirectURIs**.
- ❹ The **grantMethod** is used to determine what action to take when this client requests tokens and has not yet been granted access by the user. Uses the same values seen in Grant Options.

4.1.4.2. Service Accounts as OAuth Clients

A service account can be used as a constrained form of OAuth client. Service accounts can only request a subset of scopes that allow access to some basic user information and role-based power inside of the service account's own namespace:

- **user:info**
- **user:check-access**
- **role:<any_role>:<serviceaccount_namespace>**
- **role:<any_role>:<serviceaccount_namespace>:!**

When using a service account as an OAuth client:

- **client_id** is **system:serviceaccount:<serviceaccount_namespace>:<serviceaccount_name>**.
- **client_secret** can be any of the API tokens for that service account. For example:

```
$ oc sa get-token <serviceaccount_name>
```

- To get **WWW-Authenticate** challenges, set an **serviceaccounts.openshift.io/oauth-want-challenges** annotation on the service account to **true**.
- **redirect_uri** must match an annotation on the service account. [Redirect URIs for Service Accounts as OAuth Clients](#) provides more information.

4.1.4.3. Redirect URIs for Service Accounts as OAuth Clients

Annotation keys must have the prefix **serviceaccounts.openshift.io/oauth-redirecturi.** or **serviceaccounts.openshift.io/oauth-redirectreference.** such as:

```
serviceaccounts.openshift.io/oauth-redirecturi.<name>
```

In its simplest form, the annotation can be used to directly specify valid redirect URIs. For example:

```
"serviceaccounts.openshift.io/oauth-redirecturi.first":
"https://example.com"
"serviceaccounts.openshift.io/oauth-redirecturi.second":
"https://other.com"
```

The **first** and **second** postfixes in the above example are used to separate the two valid redirect URIs.

In more complex configurations, static redirect URIs may not be enough. For example, perhaps you want all ingresses for a route to be considered valid. This is where dynamic redirect URIs via the **serviceaccounts.openshift.io/oauth-redirectreference.** prefix come into play.

For example:

```
"serviceaccounts.openshift.io/oauth-redirectreference.first": "
{"kind\":\"OAuthRedirectReference\",\"apiVersion\":\"v1\",\"reference\":
{"kind\":\"Route\",\"name\":\"jenkins\"}}"
```

Since the value for this annotation contains serialized JSON data, it is easier to see in an expanded format:

```
{
  "kind": "OAuthRedirectReference",
  "apiVersion": "v1",
  "reference": {
    "kind": "Route",
    "name": "jenkins"
  }
}
```

Now you can see that an **OAuthRedirectReference** allows us to reference the route named **jenkins**. Thus, all ingresses for that route will now be considered valid. The full specification for an **OAuthRedirectReference** is:

```
{
  "kind": "OAuthRedirectReference",
```

```

    "apiVersion": "v1",
    "reference": {
      "kind": ..., ❶
      "name": ..., ❷
      "group": ... ❸
    }
  }
}

```

- ❶ **kind** refers to the type of the object being referenced. Currently, only **route** is supported.
- ❷ **name** refers to the name of the object. The object must be in the same namespace as the service account.
- ❸ **group** refers to the group of the object. Leave this blank, as the group for a route is the empty string.

Both annotation prefixes can be combined to override the data provided by the reference object. For example:

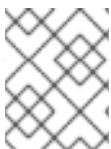
```

"serviceaccounts.openshift.io/oauth-redirecturi.first": "custompath"
"serviceaccounts.openshift.io/oauth-redirectreference.first": "
{"kind\":\"OAuthRedirectReference\",\"apiVersion\":\"v1\",\"reference\":
{"kind\":\"Route\",\"name\":\"jenkins\"}}"

```

The **first** postfix is used to tie the annotations together. Assuming that the **jenkins** route had an ingress of **https://example.com**, now **https://example.com/custompath** is considered valid, but **https://example.com** is not. The format for partially supplying override data is as follows:

Type	Syntax
Scheme	"https://"
Hostname	"//website.com"
Port	"//:8000"
Path	"examplepath"



NOTE

Specifying a host name override will replace the host name data from the referenced object, which is not likely to be desired behavior.

Any combination of the above syntax can be combined using the following format:

<scheme:>//<hostname>:<port>/<path>

The same object can be referenced more than once for more flexibility:

```

"serviceaccounts.openshift.io/oauth-redirecturi.first": "custompath"
"serviceaccounts.openshift.io/oauth-redirectreference.first": "

```

```
{\"kind\": \"OAuthRedirectReference\", \"apiVersion\": \"v1\", \"reference\": {\"kind\": \"Route\", \"name\": \"jenkins\"}}\"
\"serviceaccounts.openshift.io/oauth-redirecturi.second\": \"//:8000\"
\"serviceaccounts.openshift.io/oauth-redirectreference.second\": \"
{\"kind\": \"OAuthRedirectReference\", \"apiVersion\": \"v1\", \"reference\": {\"kind\": \"Route\", \"name\": \"jenkins\"}}\"
```

Assuming that the route named **jenkins** has an ingress of **https://example.com**, then both **https://example.com:8000** and **https://example.com/custompath** are considered valid.

Static and dynamic annotations can be used at the same time to achieve the desired behavior:

```
\"serviceaccounts.openshift.io/oauth-redirectreference.first\": \"
{\"kind\": \"OAuthRedirectReference\", \"apiVersion\": \"v1\", \"reference\": {\"kind\": \"Route\", \"name\": \"jenkins\"}}\"
\"serviceaccounts.openshift.io/oauth-redirecturi.second\": \"https://other.com\"
```

4.1.4.3.1. API Events for OAuth

In some cases the API server returns an **unexpected condition** error message that is difficult to debug without direct access to the API master log. The underlying reason for the error is purposely obscured in order to avoid providing an unauthenticated user with information about the server's state.

A subset of these errors is related to service account OAuth configuration issues. These issues are captured in events that can be viewed by non-administrator users. When encountering an **unexpected condition** server error during OAuth, run **oc get events** to view these events under **ServiceAccount**.

The following example warns of a service account that is missing a proper OAuth redirect URI:

```
$ oc get events | grep ServiceAccount
1m          1m          1          proxy          ServiceAccount
Warning    NoSAOAuthRedirectURIs  service-account-oauth-client-getter
system:serviceaccount:myproject:proxy has no redirectURIs; set
serviceaccounts.openshift.io/oauth-redirecturi.<some-value>=<redirect> or
create a dynamic URI using serviceaccounts.openshift.io/oauth-
redirectreference.<some-value>=<reference>
```

Running **oc describe sa/<service-account-name>** reports any OAuth events associated with the given service account name.

```
$ oc describe sa/proxy | grep -A5 Events
Events:
  FirstSeen      LastSeen        Count   From              Message
  SubObjectPath  Type            Reason
  -----
  -----
  3m             3m             1       service-account-oauth-client-getter
Warning        NoSAOAuthRedirectURIs
system:serviceaccount:myproject:proxy has no redirectURIs; set
serviceaccounts.openshift.io/oauth-redirecturi.<some-value>=<redirect> or
create a dynamic URI using serviceaccounts.openshift.io/oauth-
redirectreference.<some-value>=<reference>
```

The following is a list of the possible event errors:

No redirect URI annotations or an invalid URI is specified

Reason	Message
NoSAOAuthRedirectURIs	system:serviceaccount:myproject:proxy has no redirectURIs; set serviceaccounts.openshift.io/oauth-redirecturi.<some-value>=<redirect> or create a dynamic URI using serviceaccounts.openshift.io/oauth-redirectreference.<some-value>=<reference>

Invalid route specified

Reason	Message
NoSAOAuthRedirectURIs	[routes.route.openshift.io "<name>" not found, system:serviceaccount:myproject:proxy has no redirectURIs; set serviceaccounts.openshift.io/oauth-redirecturi.<some-value>=<redirect> or create a dynamic URI using serviceaccounts.openshift.io/oauth-redirectreference.<some-value>=<reference>]

Invalid reference type specified

Reason	Message
NoSAOAuthRedirectURIs	[no kind "<name>" is registered for version "v1", system:serviceaccount:myproject:proxy has no redirectURIs; set serviceaccounts.openshift.io/oauth-redirecturi.<some-value>=<redirect> or create a dynamic URI using serviceaccounts.openshift.io/oauth-redirectreference.<some-value>=<reference>]

Missing SA tokens

Reason	Message
NoSAOAuthTokens	system:serviceaccount:myproject:proxy has no tokens

4.1.4.3.1.1. Sample API Event Caused by a Possible Misconfiguration

The following steps represent one way a user could get into a broken state and how to debug or fix the issue:

1. Create a project utilizing a service account as an OAuth client.
 - a. Create YAML for a proxy service account object and ensure it uses the route **proxy**:

```
vi serviceaccount.yaml
```

Add the following sample code:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: proxy
  annotations:
```

```
    serviceaccounts.openshift.io/oauth-redirectreference.primary:
'{"kind": "OAuthRedirectReference", "apiVersion": "v1", "reference":
{"kind": "Route", "name": "proxy"}}'
```

- b. Create YAML for a route object to create a secure connection to the proxy:

```
vi route.yaml
```

Add the following sample code:

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: proxy
spec:
  to:
    name: proxy
  tls:
    termination: Reencrypt
apiVersion: v1
kind: Service
metadata:
  name: proxy
  annotations:
    service.alpha.openshift.io/serving-cert-secret-name: proxy-
    tls
spec:
  ports:
    - name: proxy
      port: 443
      targetPort: 8443
  selector:
    app: proxy
```

- c. Create a YAML for a deployment configuration to launch a proxy as a sidecar:

```
vi proxysidecar.yaml
```

Add the following sample code:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: proxy
spec:
  replicas: 1
  selector:
    matchLabels:
      app: proxy
  template:
    metadata:
      labels:
        app: proxy
    spec:
      serviceAccountName: proxy
```

```

containers:
- name: oauth-proxy
  imagePullPolicy: IfNotPresent
  ports:
  - containerPort: 8443
    name: public
  args:
  - --https-address=:8443
  - --provider=openshift
  - --openshift-service-account=proxy
  - --upstream=http://localhost:8080
  - --tls-cert=/etc/tls/private/tls.crt
  - --tls-key=/etc/tls/private/tls.key
  - --cookie-secret=SECRET
  volumeMounts:
  - mountPath: /etc/tls/private
    name: proxy-tls

- name: app
  image: openshift/hello-openshift:latest
volumes:
- name: proxy-tls
  secret:
    secretName: proxy-tls

```

d. Create the objects

```

oc create -f serviceaccount.yaml
oc create -f route.yaml
oc create -f proxysidecar.yaml

```

2. Run **oc edit sa/proxy** to edit the service account and change the **serviceaccounts.openshift.io/oauth-redirectreference** annotation to point to a Route that does not exist.

```

apiVersion: v1
imagePullSecrets:
- name: proxy-dockercfg-08d5n
kind: ServiceAccount
metadata:
  annotations:
    serviceaccounts.openshift.io/oauth-redirectreference.primary:
    '{"kind":"OAuthRedirectReference","apiVersion":"v1","reference":
    {"kind":"Route","name":"notexist"}}'
  ...

```

3. Review the OAuth log for the service to locate the server error:

```

The authorization server encountered an unexpected condition that
prevented it from fulfilling the request.

```

4. Run **oc get events** to view the **ServiceAccount** event:

```

oc get events | grep ServiceAccount

```

```

23m          23m          1          proxy          Warning
ServiceAccount
NoSAOAuthRedirectURIs    service-account-oauth-client-getter
[routes.route.openshift.io "notexist" not found,
system:serviceaccount:myproject:proxy has no redirectURIs; set
serviceaccounts.openshift.io/oauth-redirecturi.<some-value>=
<redirect> or create a dynamic URI using
serviceaccounts.openshift.io/oauth-redirectreference.<some-value>=
<reference>]

```

4.1.4.4. Integrations

All requests for OAuth tokens involve a request to **<master>/oauth/authorize**. Most authentication integrations place an authenticating proxy in front of this endpoint, or configure OpenShift Dedicated to validate credentials against a backing identity provider. Requests to **<master>/oauth/authorize** can come from user-agents that cannot display interactive login pages, such as the CLI. Therefore, OpenShift Dedicated supports authenticating using a **WWW-Authenticate** challenge in addition to interactive login flows.

If an authenticating proxy is placed in front of the **<master>/oauth/authorize** endpoint, it should send unauthenticated, non-browser user-agents **WWW-Authenticate** challenges, rather than displaying an interactive login page or redirecting to an interactive login flow.



NOTE

To prevent cross-site request forgery (CSRF) attacks against browser clients, Basic authentication challenges should only be sent if a **X-CSRF-Token** header is present on the request. Clients that expect to receive Basic **WWW-Authenticate** challenges should set this header to a non-empty value.

If the authenticating proxy cannot support **WWW-Authenticate** challenges, or if OpenShift Dedicated is configured to use an identity provider that does not support **WWW-Authenticate** challenges, users can visit **<master>/oauth/token/request** using a browser to obtain an access token manually.

4.1.4.5. OAuth Server Metadata

Applications running in OpenShift Dedicated may need to discover information about the built-in OAuth server. For example, they may need to discover what the address of the **<master>** server is without manual configuration. To aid in this, OpenShift Dedicated implements the IETF [OAuth 2.0 Authorization Server Metadata](#) draft specification.

Thus, any application running inside the cluster can issue a **GET** request to **https://openshift.default.svc/.well-known/oauth-authorization-server** to fetch the following information:

```

{
  "issuer": "https://<master>", 1
  "authorization_endpoint": "https://<master>/oauth/authorize", 2
  "token_endpoint": "https://<master>/oauth/token", 3
  "scopes_supported": [ 4
    "user:full",
    "user:info",

```

```

    "user:check-access",
    "user:list-scoped-projects",
    "user:list-projects"
  ],
  "response_types_supported": [ 5
    "code",
    "token"
  ],
  "grant_types_supported": [ 6
    "authorization_code",
    "implicit"
  ],
  "code_challenge_methods_supported": [ 7
    "plain",
    "S256"
  ]
}

```

- 1 The authorization server's issuer identifier, which is a URL that uses the **https** scheme and has no query or fragment components. This is the location where **.well-known RFC 5785** resources containing information about the authorization server are published.
- 2 URL of the authorization server's authorization endpoint. See [RFC 6749](#).
- 3 URL of the authorization server's token endpoint. See [RFC 6749](#).
- 4 JSON array containing a list of the OAuth 2.0 [RFC 6749](#) scope values that this authorization server supports. Note that not all supported scope values are advertised.
- 5 JSON array containing a list of the OAuth 2.0 **response_type** values that this authorization server supports. The array values used are the same as those used with the **response_types** parameter defined by "OAuth 2.0 Dynamic Client Registration Protocol" in [RFC 7591](#).
- 6 JSON array containing a list of the OAuth 2.0 grant type values that this authorization server supports. The array values used are the same as those used with the **grant_types** parameter defined by **OAuth 2.0 Dynamic Client Registration Protocol** in [RFC 7591](#).
- 7 JSON array containing a list of PKCE [RFC 7636](#) code challenge methods supported by this authorization server. Code challenge method values are used in the **code_challenge_method** parameter defined in [Section 4.3 of RFC 7636](#). The valid code challenge method values are those registered in the IANA **PKCE Code Challenge Methods** registry. See [IANA OAuth Parameters](#).

4.1.4.6. Obtaining OAuth Tokens

The OAuth server supports standard [authorization code grant](#) and the [implicit grant](#) OAuth authorization flows.

When requesting an OAuth token using the implicit grant flow (**response_type=token**) with a `client_id` configured to request **WWW-Authenticate challenges** (like **openshift-challenging-client**), these are the possible server responses from `/oauth/authorize`, and how they should be handled:

Status	Content	Client response
302	Location header containing an access_token parameter in the URL fragment (RFC 4.2.2)	Use the access_token value as the OAuth token
302	Location header containing an error query parameter (RFC 4.1.2.1)	Fail, optionally surfacing the error (and optional error_description) query values to the user
302	Other Location header	Follow the redirect, and process the result using these rules
401	WWW-Authenticate header present	Respond to challenge if type is recognized (e.g. Basic , Negotiate , etc), resubmit request, and process the result using these rules
401	WWW-Authenticate header missing	No challenge authentication is possible. Fail and show response body (which might contain links or details on alternate methods to obtain an OAuth token)
Other	Other	Fail, optionally surfacing response body to the user

4.2. AUTHORIZATION

4.2.1. Overview

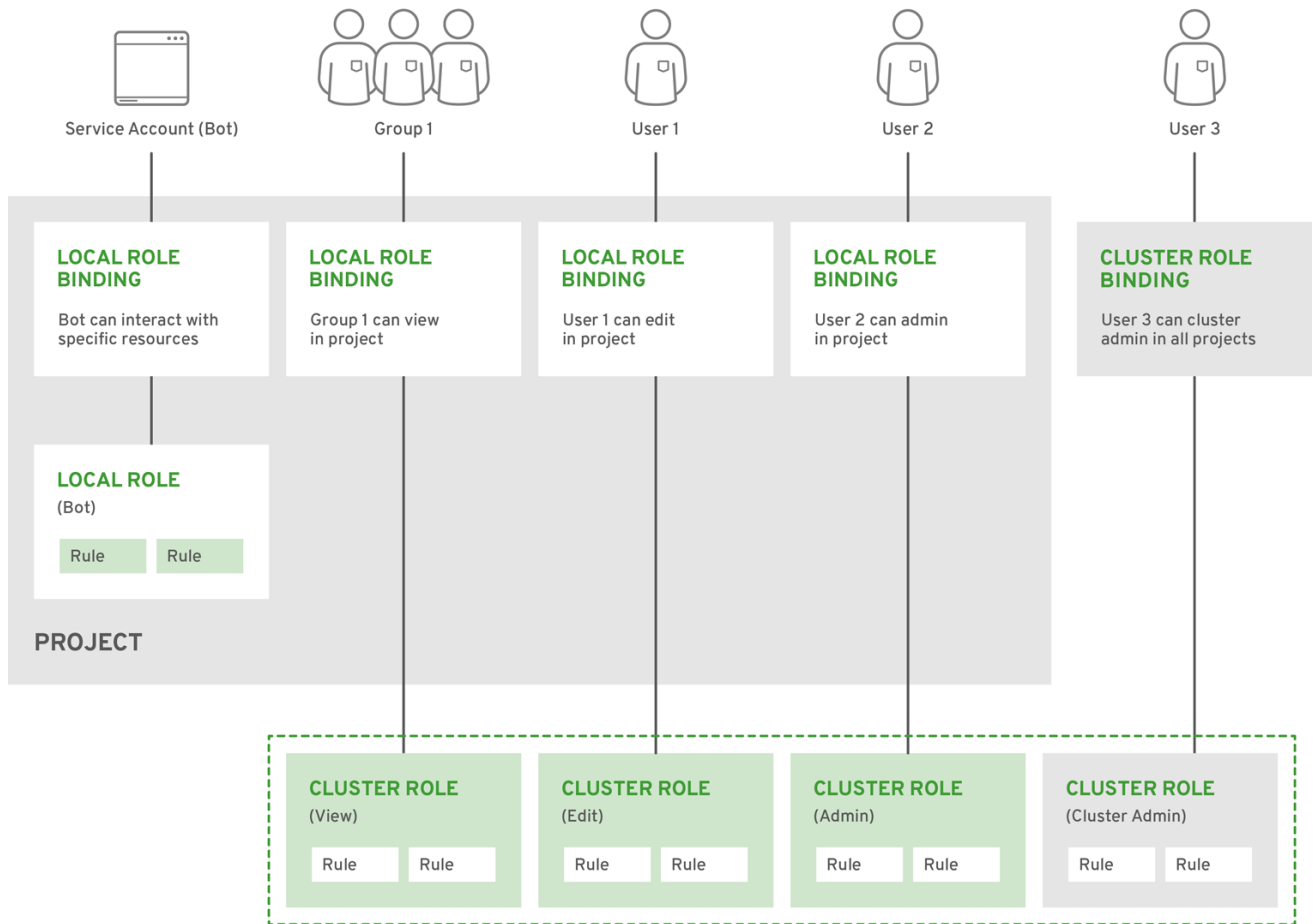
Role-based Access Control (RBAC) objects determine whether a user is allowed to perform a given [action](#) within a project.

It allows developers to use [local roles and bindings](#) to control who has access to their [projects](#). Note that authorization is a separate step from [authentication](#), which is more about determining the identity of who is taking the action.

Authorization is managed using:

Rules	Sets of permitted verbs on a set of objects . For example, whether something can create pods.
Roles	Collections of rules. Users and groups can be associated with, or <i>bound</i> to, multiple roles at the same time.
Bindings	Associations between users and/or groups with a role .

The relationships between cluster roles, local roles, cluster role bindings, local role bindings, users, groups and service accounts are illustrated below.



OPENSIFT_415489_0218

4.2.2. Evaluating Authorization

Several factors are combined to make the decision when OpenShift Dedicated evaluates authorization:

Identity	In the context of authorization, both the user name and list of groups the user belongs to.						
Action	<div>The action being performed. In most cases, this consists of:<table><tr><td>Project</td><td>The project being accessed.</td></tr><tr><td>Verb</td><td>Can be get, list, create, update, delete, deletecollection or watch.</td></tr><tr><td>Resource Name</td><td>The API endpoint being accessed.</td></tr></table></div>	Project	The project being accessed.	Verb	Can be get , list , create , update , delete , deletecollection or watch .	Resource Name	The API endpoint being accessed.
Project	The project being accessed.						
Verb	Can be get , list , create , update , delete , deletecollection or watch .						
Resource Name	The API endpoint being accessed.						
Bindings	The full list of bindings .						

OpenShift Dedicated evaluates authorizations using the following steps:

1. The identity and the project-scoped action is used to find all bindings that apply to the user or their groups.
2. Bindings are used to locate all the roles that apply.
3. Roles are used to find all the rules that apply.
4. The action is checked against each rule to find a match.
5. If no matching rule is found, the action is then denied by default.

4.2.3. Cluster and Local RBAC

There are two levels of RBAC roles and bindings that control authorization:

Cluster RBAC	Roles and bindings that are applicable across all projects. Roles that exist cluster-wide are considered <i>cluster roles</i> . Cluster role bindings can only reference cluster roles.
Local RBAC	Roles and bindings that are scoped to a given project. Roles that exist only in a project are considered <i>local roles</i> . Local role bindings can reference both cluster and local roles.

This two-level hierarchy allows re-usability over multiple projects through the cluster roles while allowing customization inside of individual projects through local roles.

During evaluation, both the cluster role bindings and the local role bindings are used. For example:

1. Cluster-wide "allow" rules are checked.
2. Locally-bound "allow" rules are checked.
3. Deny by default.

4.2.4. Cluster Roles and Local Roles

Roles are collections of policy [rules](#), which are sets of permitted verbs that can be performed on a set of resources. OpenShift Dedicated includes a set of default cluster roles that can be bound to users and groups [cluster wide](#) or [locally](#).

Default Cluster Role	Description
admin	A project manager. If used in a local binding , an admin user will have rights to view any resource in the project and modify any resource in the project except for quota.
basic-user	A user that can get basic information about projects and users.
cluster-admin	A super-user that can perform any action in any project. When bound to a user with a local binding , they have full control over quota and every action on every resource in the project.

Default Cluster Role	Description
cluster-status	A user that can get basic cluster status information.
edit	A user that can modify most objects in a project, but does not have the power to view or modify roles or bindings.
self-provisioner	A user that can create their own projects.
view	A user who cannot make any modifications, but can see most objects in a project. They cannot view or modify roles or bindings.

TIP

Remember that [users and groups](#) can be associated with, or *bound* to, multiple roles at the same time.

Project administrators can visualize roles, including a matrix of the verbs and resources each are associated using the CLI to view local bindings.

**IMPORTANT**

The cluster role bound to the project administrator is limited in a project via a [local binding](#). It is not bound [cluster-wide](#) like the cluster roles granted to the **cluster-admin** or **system:admin**.

Cluster roles are [roles](#) defined at the cluster level, but can be bound either at the cluster level or at the project level.

4.2.5. Security Context Constraints

In addition to the [RBAC resources](#) that control what a user can do, OpenShift Dedicated provides *security context constraints* (SCC) that control the actions that a [pod](#) can perform and what it has the ability to access. Administrators can [manage SCCs](#) using the CLI.

SCCs are also very useful for managing access to persistent storage.

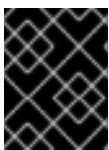
SCCs are objects that define a set of conditions that a pod must run with in order to be accepted into the system. They allow an administrator to control the following:

1. Running of privileged containers.
2. Capabilities a container can request to be added.
3. Use of host directories as volumes.
4. The SELinux context of the container.
5. The user ID.
6. The use of host namespaces and networking.
7. Allocating an **FSGroup** that owns the pod's volumes

8. Configuring allowable supplemental groups
9. Requiring the use of a read only root file system
10. Controlling the usage of volume types
11. Configuring allowable seccomp profiles

Seven SCCs are added to the cluster by default, and are viewable by cluster administrators using the CLI:

```
$ oc get scc
NAME          PRIV          CAPS          SELINUX        RUNASUSER
FSGROUP       SUPGROUP      PRIORITY      READONLYROOTFS  VOLUMES
anyuid        false        []            MustRunAs      RunAsAny
RunAsAny      RunAsAny      10            false          [configMap
downwardAPI   emptyDir      persistentVolumeClaim secret]
hostaccess    false        []            MustRunAs      MustRunAsRange
MustRunAs     RunAsAny      <none>        false          [configMap
downwardAPI   emptyDir      hostPath      persistentVolumeClaim secret]
hostmount-anyuid false        []            MustRunAs      RunAsAny
RunAsAny      RunAsAny      <none>        false          [configMap
downwardAPI   emptyDir      hostPath      nfs persistentVolumeClaim secret]
hostnetwork   false        []            MustRunAs      MustRunAsRange
MustRunAs     MustRunAs     <none>        false          [configMap
downwardAPI   emptyDir      persistentVolumeClaim secret]
nonroot       false        []            MustRunAs      MustRunAsNonRoot
RunAsAny      RunAsAny      <none>        false          [configMap
downwardAPI   emptyDir      persistentVolumeClaim secret]
privileged    true         [*]           RunAsAny       RunAsAny
RunAsAny      RunAsAny      <none>        false          [*]
restricted    false        []            MustRunAs      MustRunAsRange
MustRunAs     RunAsAny      <none>        false          [configMap
downwardAPI   emptyDir      persistentVolumeClaim secret]
```



IMPORTANT

Do not modify the default SCCs. Customizing the default SCCs can lead to issues when OpenShift Dedicated is upgraded.

The definition for each SCC is also viewable by cluster administrators using the CLI. For example, for the privileged SCC:

```
# oc export scc/privileged
allowHostDirVolumePlugin: true
allowHostIPC: true
allowHostNetwork: true
allowHostPID: true
allowHostPorts: true
allowPrivilegedContainer: true
allowedCapabilities: ❶
- '*'
apiVersion: v1
defaultAddCapabilities: [] ❷
```

```

fsGroup: ❸
  type: RunAsAny
groups: ❹
- system:cluster-admins
- system:nodes
kind: SecurityContextConstraints
metadata:
  annotations:
    kubernetes.io/description: 'privileged allows access to all privileged
and host
    features and the ability to run as any user, any group, any fsGroup,
and with
    any SELinux context. WARNING: this is the most relaxed SCC and
should be used
    only for cluster administration. Grant with caution.'
  creationTimestamp: null
  name: privileged
priority: null
readOnlyRootFilesystem: false
requiredDropCapabilities: [] ❺
runAsUser: ❻
  type: RunAsAny
seLinuxContext: ❼
  type: RunAsAny
seccompProfiles:
- '*'
supplementalGroups: ❽
  type: RunAsAny
users: ❾
- system:serviceaccount:default:registry
- system:serviceaccount:default:router
- system:serviceaccount:openshift-infra:build-controller
volumes:
- '*'

```

- ❶ A list of capabilities that can be requested by a pod. An empty list means that none of capabilities can be requested while the special symbol * allows any capabilities.
- ❷ A list of additional capabilities that will be added to any pod.
- ❸ The **FSGroup** strategy which dictates the allowable values for the Security Context.
- ❹ The groups that have access to this SCC.
- ❺ A list of capabilities that will be dropped from a pod.
- ❻ The run as user strategy type which dictates the allowable values for the Security Context.
- ❼ The SELinux context strategy type which dictates the allowable values for the Security Context.
- ❽ The supplemental groups strategy which dictates the allowable supplemental groups for the Security Context.
- ❾ The users who have access to this SCC.

The **users** and **groups** fields on the SCC control which SCCs can be used. By default, cluster administrators, nodes, and the build controller are granted access to the privileged SCC. All authenticated users are granted access to the restricted SCC.

Docker has a [default list of capabilities](#) that are allowed for each container of a pod. The containers use the capabilities from this default list, but pod manifest authors can alter it by requesting additional capabilities or dropping some of defaulting. The **allowedCapabilities**, **defaultAddCapabilities**, and **requiredDropCapabilities** fields are used to control such requests from the pods, and to dictate which capabilities can be requested, which ones must be added to each container, and which ones must be forbidden.

The privileged SCC:

- allows privileged pods.
- allows host directories to be mounted as volumes.
- allows a pod to run as any user.
- allows a pod to run with any MCS label.
- allows a pod to use the host's IPC namespace.
- allows a pod to use the host's PID namespace.
- allows a pod to use any FSGroup.
- allows a pod to use any supplemental group.
- allows a pod to use any seccomp profiles.
- allows a pod to request any capabilities.

The restricted SCC:

- ensures pods cannot run as privileged.
- ensures pods cannot use host directory volumes.
- requires that a pod run as a user in a pre-allocated range of UIDs.
- requires that a pod run with a pre-allocated MCS label.
- allows a pod to use any FSGroup.
- allows a pod to use any supplemental group.



NOTE

For more information about each SCC, see the **kubernetes.io/description** annotation available on the SCC.

SCCs are comprised of settings and strategies that control the security features a pod has access to. These settings fall into three categories:

Controlled by a boolean	Fields of this type default to the most restrictive value. For example, AllowPrivilegedContainer is always set to false if unspecified.
Controlled by an allowable set	Fields of this type are checked against the set to ensure their value is allowed.
Controlled by a strategy	Items that have a strategy to generate a value provide: <ul style="list-style-type: none"> • A mechanism to generate the value, and • A mechanism to ensure that a specified value falls into the set of allowable values.

4.2.5.1. SCC Strategies

4.2.5.1.1. RunAsUser

1. **MustRunAs** - Requires a **runAsUser** to be configured. Uses the configured **runAsUser** as the default. Validates against the configured **runAsUser**.
2. **MustRunAsRange** - Requires minimum and maximum values to be defined if not using pre-allocated values. Uses the minimum as the default. Validates against the entire allowable range.
3. **MustRunAsNonRoot** - Requires that the pod be submitted with a non-zero **runAsUser** or have the **USER** directive defined in the image. No default provided.
4. **RunAsAny** - No default provided. Allows any **runAsUser** to be specified.

4.2.5.1.2. SELinuxContext

1. **MustRunAs** - Requires **seLinuxOptions** to be configured if not using pre-allocated values. Uses **seLinuxOptions** as the default. Validates against **seLinuxOptions**.
2. **RunAsAny** - No default provided. Allows any **seLinuxOptions** to be specified.

4.2.5.1.3. SupplementalGroups

1. **MustRunAs** - Requires at least one range to be specified if not using pre-allocated values. Uses the minimum value of the first range as the default. Validates against all ranges.
2. **RunAsAny** - No default provided. Allows any **supplementalGroups** to be specified.

4.2.5.1.4. FSGroup

1. **MustRunAs** - Requires at least one range to be specified if not using pre-allocated values. Uses the minimum value of the first range as the default. Validates against the first ID in the first range.
2. **RunAsAny** - No default provided. Allows any **fsGroup** ID to be specified.

4.2.5.2. Controlling Volumes

The usage of specific volume types can be controlled by setting the **volumes** field of the SCC. The allowable values of this field correspond to the volume sources that are defined when creating a volume:

- [azureFile](#)
- [azureDisk](#)
- [flocker](#)
- [flexVolume](#)
- [hostPath](#)
- [emptyDir](#)
- [gcePersistentDisk](#)
- [awsElasticBlockStore](#)
- [gitRepo](#)
- [secret](#)
- [nfs](#)
- [iscsi](#)
- [glusterfs](#)
- [persistentVolumeClaim](#)
- [rbd](#)
- [cinder](#)
- [cephFS](#)
- [downwardAPI](#)
- [fc](#)
- [configMap](#)
- [vsphereVolume](#)
- [quobyte](#)
- [photonPersistentDisk](#)
- [projected](#)
- [portworxVolume](#)
- [scaleIO](#)
- [storageos](#)

- `*` (a special value to allow the use of all volume types)
- **none** (a special value to disallow the use of all volumes types. Exist only for backwards compatibility)

The recommended minimum set of allowed volumes for new SCCs are **configMap**, **downwardAPI**, **emptyDir**, **persistentVolumeClaim**, **secret**, and **projected**.



NOTE

The list of allowable volume types is not exhaustive because new types are added with each release of OpenShift Dedicated.



NOTE

For backwards compatibility, the usage of **allowHostDirVolumePlugin** overrides settings in the **volumes** field. For example, if **allowHostDirVolumePlugin** is set to false but allowed in the **volumes** field, then the **hostPath** value will be removed from **volumes**.

4.2.5.3. Restricting Access to FlexVolumes

OpenShift Dedicated provides additional control of FlexVolumes based on their driver. When SCC allows the usage of FlexVolumes, pods can request any FlexVolumes. However, when the cluster administrator specifies driver names in the **AllowedFlexVolumes** field, pods must only use FlexVolumes with these drivers.

Example of Limiting Access to Only Two FlexVolumes

```
volumes:
- flexVolume
allowedFlexVolumes:
- driver: example/lvm
- driver: example/cifs
```

4.2.5.4. Seccomp

SeccompProfiles lists the allowed profiles that can be set for the pod or container's seccomp annotations. An unset (nil) or empty value means that no profiles are specified by the pod or container. Use the wildcard `*` to allow all profiles. When used to generate a value for a pod, the first non-wildcard profile is used as the default.

4.2.5.5. Admission

Admission control with SCCs allows for control over the creation of resources based on the capabilities granted to a user.

In terms of the SCCs, this means that an admission controller can inspect the user information made available in the context to retrieve an appropriate set of SCCs. Doing so ensures the pod is authorized to make requests about its operating environment or to generate a set of constraints to apply to the pod.

The set of SCCs that admission uses to authorize a pod are determined by the user identity and groups that the user belongs to. Additionally, if the pod specifies a service account, the set of allowable SCCs includes any constraints accessible to the service account.

Admission uses the following approach to create the final security context for the pod:

1. Retrieve all SCCs available for use.
2. Generate field values for security context settings that were not specified on the request.
3. Validate the final settings against the available constraints.

If a matching set of constraints is found, then the pod is accepted. If the request cannot be matched to an SCC, the pod is rejected.

A pod must validate every field against the SCC. The following are examples for just two of the fields that must be validated:



NOTE

These examples are in the context of a strategy using the preallocated values.

A FSGroup SCC Strategy of MustRunAs

If the pod defines a **fsGroup** ID, then that ID must equal the default **fsGroup** ID. Otherwise, the pod is not validated by that SCC and the next SCC is evaluated.

If the **SecurityContextConstraints.fsGroup** field has value **RunAsAny** and the pod specification omits the **Pod.spec.securityContext.fsGroup**, then this field is considered valid. Note that it is possible that during validation, other SCC settings will reject other pod fields and thus cause the pod to fail.

A SupplementalGroups SCC Strategy of MustRunAs

If the pod specification defines one or more **supplementalGroups** IDs, then the pod's IDs must equal one of the IDs in the namespace's **openshift.io/sa.scc.supplemental-groups** annotation. Otherwise, the pod is not validated by that SCC and the next SCC is evaluated.

If the **SecurityContextConstraints.supplementalGroups** field has value **RunAsAny** and the pod specification omits the **Pod.spec.securityContext.supplementalGroups**, then this field is considered valid. Note that it is possible that during validation, other SCC settings will reject other pod fields and thus cause the pod to fail.

4.2.5.5.1. SCC Prioritization

SCCs have a priority field that affects the ordering when attempting to validate a request by the admission controller. A higher priority SCC is moved to the front of the set when sorting. When the complete set of available SCCs are determined they are ordered by:

1. Highest priority first, nil is considered a 0 priority
2. If priorities are equal, the SCCs will be sorted from most restrictive to least restrictive
3. If both priorities and restrictions are equal the SCCs will be sorted by name

By default, the anyuid SCC granted to cluster administrators is given priority in their SCC set. This allows cluster administrators to run pods as any user by without specifying a **RunAsUser** on the pod's **SecurityContext**. The administrator may still specify a **RunAsUser** if they wish.

4.2.5.5.2. Understanding Pre-allocated Values and Security Context Constraints

The admission controller is aware of certain conditions in the security context constraints that trigger it to look up pre-allocated values from a namespace and populate the security context constraint before processing the pod. Each SCC strategy is evaluated independently of other strategies, with the pre-allocated values (where allowed) for each policy aggregated with pod specification values to make the final values for the various IDs defined in the running pod.

The following SCCs cause the admission controller to look for pre-allocated values when no ranges are defined in the pod specification:

1. A **RunAsUser** strategy of **MustRunAsRange** with no minimum or maximum set. Admission looks for the **openshift.io/sa.scc.uid-range** annotation to populate range fields.
2. An **SELinuxContext** strategy of **MustRunAs** with no level set. Admission looks for the **openshift.io/sa.scc.mcs** annotation to populate the level.
3. A **FSGroup** strategy of **MustRunAs**. Admission looks for the **openshift.io/sa.scc.supplemental-groups** annotation.
4. A **SupplementalGroups** strategy of **MustRunAs**. Admission looks for the **openshift.io/sa.scc.supplemental-groups** annotation.

During the generation phase, the security context provider will default any values that are not specifically set in the pod. Defaulting is based on the strategy being used:

1. **RunAsAny** and **MustRunAsNonRoot** strategies do not provide default values. Thus, if the pod needs a field defined (for example, a group ID), this field must be defined inside the pod specification.
2. **MustRunAs** (single value) strategies provide a default value which is always used. As an example, for group IDs: even if the pod specification defines its own ID value, the namespace's default field will also appear in the pod's groups.
3. **MustRunAsRange** and **MustRunAs** (range-based) strategies provide the minimum value of the range. As with a single value **MustRunAs** strategy, the namespace's default value will appear in the running pod. If a range-based strategy is configurable with multiple ranges, it will provide the minimum value of the first configured range.



NOTE

FSGroup and **SupplementalGroups** strategies fall back to the **openshift.io/sa.scc.uid-range** annotation if the **openshift.io/sa.scc.supplemental-groups** annotation does not exist on the namespace. If neither exist, the SCC will fail to create.



NOTE

By default, the annotation-based **FSGroup** strategy configures itself with a single range based on the minimum value for the annotation. For example, if your annotation reads **1/3**, the **FSGroup** strategy will configure itself with a minimum and maximum of **1**. If you want to allow more groups to be accepted for the **FSGroup** field, you can configure a custom SCC that does not use the annotation.

**NOTE**

The **openshift.io/sa.scc.supplemental-groups** annotation accepts a comma delimited list of blocks in the format of **<start>/<length>** or **<start>-<end>**. The **openshift.io/sa.scc.uid-range** annotation accepts only a single block.

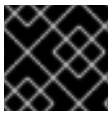
4.3. PERSISTENT STORAGE

4.3.1. Overview

Managing storage is a distinct problem from managing compute resources. OpenShift Dedicated leverages the Kubernetes persistent volume (PV) framework to allow administrators to provision persistent storage for a cluster. Using persistent volume claims (PVCs), developers can request PV resources without having specific knowledge of the underlying storage infrastructure.

PVCs are specific to a [project](#) and are created and used by developers as a means to use a PV. PV resources on their own are not scoped to any single project; they can be shared across the entire OpenShift Dedicated cluster and claimed from any project. After a PV has been [bound](#) to a PVC, however, that PV cannot then be bound to additional PVCs. This has the effect of scoping a bound PV to a single [namespace](#) (that of the binding project).

PVs are defined by a **PersistentVolume** API object, which represents a piece of existing networked storage in the cluster that has been provisioned by an administrator. It is a resource in the cluster just like a node is a cluster resource. PVs are volume plug-ins like **Volumes**, but have a lifecycle independent of any individual [pod](#) that uses the PV. PV objects capture the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

**IMPORTANT**

High availability of storage in the infrastructure is left to the underlying storage provider.

PVCs are defined by a **PersistentVolumeClaim** API object, which represents a request for storage by a developer. It is similar to a pod in that pods consume node resources and PVCs consume PV resources. For example, pods can request specific levels of resources (e.g., CPU and memory), while PVCs can request specific [storage capacity](#) and [access modes](#) (e.g., they can be mounted once read/write or many times read-only).

4.3.2. Lifecycle of a Volume and Claim

PVs are resources in the cluster. PVCs are requests for those resources and also act as claim checks to the resource. The interaction between PVs and PVCs have the following lifecycle.

4.3.2.1. Provisioning

In response to requests from a developer defined in a PVC, a cluster administrator configures one or more dynamic provisioners that provision storage and a matching PV.

Alternatively, a cluster administrator can create a number of PVs in advance, which carry the details of the real storage that is available for use by cluster users. PVs exist in the API and are available for consumption.

4.3.2.2. Binding

A user creates a **PersistentVolumeClaim** with a specific amount of storage requested and with certain access modes and optionally a **StorageClass**. A control loop in the master watches for new PVCs. It either finds a matching PV or waits for a provisioner for the **StorageClass** to create one, then binds them together.

The user will always get at least what they asked for, but the volume might be in excess of what was requested. This is especially true with manually provisioned PVs. To minimize the excess, OpenShift Dedicated binds to the smallest PV that matches all other criteria.

Claims remain unbound indefinitely if a matching volume does not exist or cannot be created with any available provisioner servicing a **StorageClass**. Claims are bound as matching volumes become available. For example, a cluster with many manually provisioned 50Gi volumes would not match a PVC requesting 100Gi. The PVC can be bound when a 100Gi PV is added to the cluster.

4.3.2.3. Using

Pods use claims as volumes. The cluster inspects the claim to find the bound volume and mounts that volume for a pod. For those volumes that support multiple access modes, the user specifies which mode is desired when using their claim as a volume in a pod.

Once a user has a claim and that claim is bound, the bound PV belongs to the user for as long as they need it. Users schedule pods and access their claimed PVs by including a **persistentVolumeClaim** in their pod's volumes block. See [below](#) for syntax details.

4.3.2.4. Releasing

When a user is done with a volume, they can delete the PVC object from the API which allows reclamation of the resource. The volume is considered "released" when the claim is deleted, but it is not yet available for another claim. The previous claimant's data remains on the volume which must be handled according to policy.

4.3.2.5. Reclaiming

The reclaim policy of a **PersistentVolume** tells the cluster what to do with the volume after it is released. Volumes reclaim policy can either be **Retained**, **Recycled**, or **Deleted**.

Retained reclaim policy allows manual reclamation of the resource for those volume plug-ins that support it. **Deleted** reclaim policy deletes both the **PersistentVolume** object from OpenShift Dedicated and the associated storage asset in external infrastructure, such as AWS EBS, GCE PD, or Cinder volume.



NOTE

Volumes that were dynamically provisioned are always deleted.

4.3.2.5.1. Recycling

If supported by appropriate volume plug-in, recycling performs a basic scrub (`rm -rf /thevolume/*`) on the volume and makes it available again for a new claim.

**WARNING**

The **recycle** reclaim policy is deprecated in favor of dynamic provisioning and it will be removed in future releases.

4.3.3. Persistent Volumes

Each PV contains a **spec** and **status**, which is the specification and status of the volume.

Persistent Volume Object Definition

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  nfs:
    path: /tmp
    server: 172.17.0.2
```

4.3.3.1. Types of Persistent Volumes

OpenShift Dedicated supports the following **PersistentVolume** plug-ins:

- NFS
- HostPath
- GlusterFS
- Ceph RBD
- OpenStack Cinder
- AWS Elastic Block Store (EBS)
- GCE Persistent Disk
- iSCSI
- Fibre Channel
- Azure Disk
- Azure File

- VMWare vSphere
- Local

4.3.3.2. Capacity

Generally, a PV has a specific storage capacity. This is set using the PV's **capacity** attribute.

Currently, storage capacity is the only resource that can be set or requested. Future attributes may include IOPS, throughput, etc.

4.3.3.3. Access Modes

A **PersistentVolume** can be mounted on a host in any way supported by the resource provider. Providers will have different capabilities and each PV's access modes are set to the specific modes supported by that particular volume. For example, NFS can support multiple read/write clients, but a specific NFS PV might be exported on the server as read-only. Each PV gets its own set of access modes describing that specific PV's capabilities.

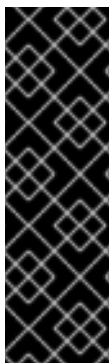
Claims are matched to volumes with similar access modes. The only two matching criteria are access modes and size. A claim's access modes represent a request. Therefore, the user may be granted more, but never less. For example, if a claim requests RWO, but the only volume available was an NFS PV (RWO+ROX+RWX), the claim would match NFS because it supports RWO.

Direct matches are always attempted first. The volume's modes must match or contain more modes than you requested. The size must be greater than or equal to what is expected. If two types of volumes (NFS and iSCSI, for example) both have the same set of access modes, then either of them will match a claim with those modes. There is no ordering between types of volumes and no way to choose one type over another.

All volumes with the same modes are grouped, then sorted by size (smallest to largest). The binder gets the group with matching modes and iterates over each (in size order) until one size matches.

The access modes are:

Access Mode	CLI Abbreviation	Description
ReadWriteOnce	RWO	The volume can be mounted as read-write by a single node.
ReadOnlyMany	ROX	The volume can be mounted read-only by many nodes.
ReadWriteMany	RWX	The volume can be mounted as read-write by many nodes.



IMPORTANT

A volume's **AccessModes** are descriptors of the volume's capabilities. They are not enforced constraints. The storage provider is responsible for runtime errors resulting from invalid use of the resource.

For example, a GCE Persistent Disk has **AccessModes ReadWriteOnce** and **ReadOnlyMany**. The user must mark their claims as **read-only** if they want to take advantage of the volume's ability for ROX. Errors in the provider show up at runtime as mount errors.

The table below lists the access modes supported by different persistent volumes:

Table 4.1. Supported Access Modes for Persistent Volumes

Volume Plug-in	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
AWS EBS		-	-
Azure File			
Azure Disk		-	-
Ceph RBD			-
Fibre Channel			-
GCE Persistent Disk		-	-
GlusterFS			
HostPath		-	-
iSCSI			-
NFS			
Openstack Cinder		-	-
VMWare vSphere		-	-
Local		-	-

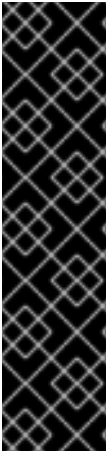


NOTE

- If pods rely on AWS EBS, GCE Persistent Disks, or Openstack Cinder PVs, use a [recreate deployment strategy](#)

4.3.3.4. OpenShift Dedicated Restrictions

The following restrictions apply when using persistent volumes with OpenShift Dedicated:



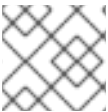
IMPORTANT

- PVs are provisioned with either EBS volumes (AWS) or GCP storage (GCP), depending on where the cluster is provisioned.
- Only RWO access mode is applicable, since EBS volumes and GCE Persistent Disks cannot be mounted to multiple nodes.
- **emptyDir** has the same lifecycle as the pod:
 - **emptyDir** volumes survive container crashes/restarts.
 - **emptyDir** volumes are deleted when the pod is deleted.

4.3.3.5. Reclaim Policy

The current reclaim policies are:

Reclaim Policy	Description
Retain	Manual reclamation
Recycle	Basic scrub (e.g, rm -rf /<volume>/*)



NOTE

Currently, only NFS and HostPath support the 'Recycle' reclaim policy.



WARNING

The **recycle** reclaim policy is deprecated in favor of dynamic provisioning and it will be removed in future releases.

4.3.3.6. Phase

Volumes can be found in one of the following phases:

Phase	Description
Available	A free resource that is not yet bound to a claim.
Bound	The volume is bound to a claim.
Released	The claim has been deleted, but the resource is not yet reclaimed by the cluster.

Phase	Description
Failed	The volume has failed its automatic reclamation.

The CLI shows the name of the PVC bound to the PV.

4.3.4. Persistent Volume Claims

Each PVC contains a **spec** and **status**, which is the specification and status of the claim.

Persistent Volume Claim Object Definition

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
  storageClassName: gold
```

4.3.4.1. Storage Class

Claims can optionally request a specific **StorageClass** by specifying its name in the **storageClassName** attribute. Only PVs of the requested class, ones with the same **storageClassName** as the PVC, can be bound to the PVC. The cluster administrator can configure dynamic provisioners to service one or more storage classes. They create a PV on demand that matches the specifications in the PVC, if they are able.

The cluster administrator can also set a default **StorageClass** for all PVCs. When a default storage class is configured, the PVC must explicitly ask for **StorageClass** or **storageClassName** annotations set to "" to get bound to a PV with a no storage class.

4.3.4.2. Access Modes

Claims use the same conventions as volumes when requesting storage with specific access modes.

4.3.4.3. Resources

Claims, like pods, can request specific quantities of a resource. In this case, the request is for storage. The same resource model applies to both volumes and claims.

4.3.4.4. Claims As Volumes

Pods access storage by using the claim as a volume. Claims must exist in the same namespace as the pod using the claim. The cluster finds the claim in the pod's namespace and uses it to get the **PersistentVolume** backing the claim. The volume is then mounted to the host and into the pod:

```

kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
  - name: myfrontend
    image: dockerfile/nginx
    volumeMounts:
    - mountPath: "/var/www/html"
      name: mypd
  volumes:
  - name: mypd
    persistentVolumeClaim:
      claimName: myclaim

```

4.4. SOURCE CONTROL MANAGEMENT

OpenShift Dedicated takes advantage of preexisting source control management (SCM) systems hosted either internally (such as an in-house Git server) or externally (for example, on [GitHub](#), [Bitbucket](#), etc.). Currently, OpenShift Dedicated only supports [Git](#) solutions.

SCM integration is tightly coupled with [builds](#), the two points being:

- Creating a **BuildConfig** using a repository, which allows building your application inside of OpenShift Dedicated. You can create a **BuildConfig** [manually](#) or let OpenShift Dedicated create it [automatically](#) by inspecting your repository.
- [Triggering a build](#) upon repository changes.

4.5. ADMISSION CONTROLLERS

4.5.1. Overview

Admission control plug-ins intercept requests to the master API prior to persistence of a resource, but after the request is authenticated and authorized.

Each admission control plug-in is run in sequence before a request is accepted into the cluster. If any plug-in in the sequence rejects the request, the entire request is rejected immediately, and an error is returned to the end-user.

Admission control plug-ins may modify the incoming object in some cases to apply system configured defaults. In addition, admission control plug-ins may modify related resources as part of request processing to do things such as incrementing quota usage.



WARNING

The OpenShift Dedicated master has a default list of plug-ins that are enabled by default for each type of resource (Kubernetes and OpenShift Dedicated). These are required for the proper functioning of the master. Modifying these lists is not recommended unless you strictly know what you are doing. Future versions of the product may use a different set of plug-ins and may change their ordering. If you do override the default list of plug-ins in the master configuration file, you are responsible for updating it to reflect requirements of newer versions of the OpenShift Dedicated master.

4.5.2. General Admission Rules

Starting in OpenShift Dedicated uses a single admission chain for Kubernetes and OpenShift Dedicated resources. This changed from and before where we had separate admission chains. This means that the top-level **admissionConfig.pluginConfig** element can now contain the admission plug-in configuration, which used to be contained in **kubernetesMasterConfig.admissionConfig.pluginConfig**.

The **kubernetesMasterConfig.admissionConfig.pluginConfig** should be moved and merged into **admissionConfig.pluginConfig**.

Also, starting in all the supported admission plug-ins are ordered in the single chain for you. You should no longer set **admissionConfig.pluginOrderOverride** or the **kubernetesMasterConfig.admissionConfig.pluginOrderOverride**. Instead, you should enable plug-ins that are off by default by either adding their plug-in-specific configuration, or adding a **DefaultAdmissionConfig** stanza like this:

```
admissionConfig:
  pluginConfig:
    AlwaysPullImages: 1
    configuration:
      kind: DefaultAdmissionConfig
      apiVersion: v1
      disable: false 2
```

- 1 Admission plug-in name.
- 2 Indicates that a plug-in should be enabled. It is optional and shown here only for reference.

Setting **disable** to **true** will disable an admission plug-in that defaults to on.

**WARNING**

Admission plug-ins are commonly used to help enforce security on the API server. Be careful when disabling them.

**NOTE**

If you were previously using **admissionConfig** elements that cannot be safely combined into a single admission chain, you will get a warning in your API server logs and your API server will start with two separate admission chains for legacy compatibility. Update your **admissionConfig** to resolve the warning.

4.6. OTHER API OBJECTS

4.6.1. LimitRange

A limit range provides a mechanism to enforce min/max limits placed on resources in a Kubernetes [namespace](#).

By adding a limit range to your namespace, you can enforce the minimum and maximum amount of CPU and Memory consumed by an individual pod or container.

4.6.2. ResourceQuota

Kubernetes can limit both the number of objects created in a [namespace](#), and the total amount of resources requested across objects in a namespace. This facilitates sharing of a single Kubernetes cluster by several teams, each in a namespace, as a mechanism of preventing one team from starving another team of cluster resources.

4.6.3. Resource

A Kubernetes **Resource** is something that can be requested by, allocated to, or consumed by a pod or container. Examples include memory (RAM), CPU, disk-time, and network bandwidth.

See the [Developer Guide](#) for more information.

4.6.4. Secret

[Secrets](#) are storage for sensitive information, such as keys, passwords, and certificates. They are accessible by the intended pod(s), but held separately from their definitions.

4.6.5. PersistentVolume

A [persistent volume](#) is an object (**PersistentVolume**) in the infrastructure provisioned by the cluster administrator. Persistent volumes provide durable storage for stateful applications.

4.6.6. PersistentVolumeClaim

A **PersistentVolumeClaim** object is a [request for storage by a pod author](#). Kubernetes matches the claim against the pool of available volumes and binds them together. The claim is then used as a volume by a pod. Kubernetes makes sure the volume is available on the same node as the pod that requires it.

4.6.6.1. Custom Resources

A *custom resource* is an extension of the Kubernetes API that extends the API or allows you to introduce your own API into a project or a cluster.

4.6.7. OAuth Objects

4.6.7.1. OAuthClient

An **OAuthClient** represents an OAuth client, as described in [RFC 6749, section 2](#).

The following **OAuthClient** objects are automatically created:

openshift-web-console	Client used to request tokens for the web console
openshift-browser-client	Client used to request tokens at /oauth/token/request with a user-agent that can handle interactive logins
openshift-challenging-client	Client used to request tokens with a user-agent that can handle WWW-Authenticate challenges

OAuthClient Object Definition

```
kind: "OAuthClient"
accessTokenMaxAgeSeconds: null ❶
apiVersion: "v1"
metadata:
  name: "openshift-web-console" ❷
  selflink: "/oapi/v1/oauthclients/openshift-web-console"
  resourceVersion: "1"
  creationTimestamp: "2015-01-01T01:01:01Z"
respondWithChallenges: false ❸
secret: "45e27750-a8aa-11e4-b2ea-3c970e4b7ffe" ❹
redirectURIs:
  - "https://localhost:8443" ❺
```

❶ The lifetime of access tokens in seconds (see [the description below](#)).

❷ The **name** is used as the **client_id** parameter in OAuth requests.

❸

When **respondWithChallenges** is set to **true**, unauthenticated requests to **/oauth/authorize** will result in **WWW-Authenticate** challenges, if supported by the configured

- 4 The value in the **secret** parameter is used as the **client_secret** parameter in an authorization code flow.
- 5 One or more absolute URIs can be placed in the **redirectURIs** section. The **redirect_uri** parameter sent with authorization requests must be prefixed by one of the specified **redirectURIs**.

The **accessTokenMaxAgeSeconds** value overrides the default **accessTokenMaxAgeSeconds** value in the master configuration file for individual OAuth clients. Setting this value for a client allows long-lived access tokens for that client without affecting the lifetime of other clients.

- If **null**, the default value in the master configuration file is used.
- If set to **0**, the token will not expire.
- If set to a value greater than **0**, tokens issued for that client are given the specified expiration time. For example, **accessTokenMaxAgeSeconds: 172800** would cause the token to expire 48 hours after being issued.

4.6.7.2. OAuthClientAuthorization

An **OAuthClientAuthorization** represents an approval by a **User** for a particular **OAuthClient** to be given an **OAuthAccessToken** with particular scopes.

Creation of **OAuthClientAuthorization** objects is done during an authorization request to the **OAuth** server.

OAuthClientAuthorization Object Definition

```
kind: "OAuthClientAuthorization"
apiVersion: "v1"
metadata:
  name: "bob:openshift-web-console"
  resourceVersion: "1"
  creationTimestamp: "2015-01-01T01:01:01-00:00"
clientName: "openshift-web-console"
userName: "bob"
userID: "9311ac33-0fde-11e5-97a1-3c970e4b7ffe"
scopes: []
```

4.6.7.3. OAuthAuthorizeToken

An **OAuthAuthorizeToken** represents an **OAuth** authorization code, as described in [RFC 6749, section 1.3.1](#).

An **OAuthAuthorizeToken** is created by a request to the **/oauth/authorize** endpoint, as described in [RFC 6749, section 4.1.1](#).

An **OAuthAuthorizeToken** can then be used to obtain an **OAuthAccessToken** with a request to the **/oauth/token** endpoint, as described in [RFC 6749, section 4.1.3](#).

OAuthAuthorizeToken Object Definition

```
kind: "OAuthAuthorizeToken"
apiVersion: "v1"
metadata:
  name: "MDAwYjM5YjMtMzM1MC00NDY4LTkxODItOTA2OTE2YzE0M2Fj" 1
  resourceVersion: "1"
  creationTimestamp: "2015-01-01T01:01:01-00:00"
clientName: "openshift-web-console" 2
expiresIn: 300 3
scopes: []
redirectURI: "https://localhost:8443/console/oauth" 4
userName: "bob" 5
userID: "9311ac33-0fde-11e5-97a1-3c970e4b7ffe" 6
```

- 1 **name** represents the token name, used as an authorization code to exchange for an OAuthAccessToken.
- 2 The **clientName** value is the OAuthClient that requested this token.
- 3 The **expiresIn** value is the expiration in seconds from the creationTimestamp.
- 4 The **redirectURI** value is the location where the user was redirected to during the authorization flow that resulted in this token.
- 5 **userName** represents the name of the User this token allows obtaining an OAuthAccessToken for.
- 6 **userID** represents the UID of the User this token allows obtaining an OAuthAccessToken for.

4.6.7.4. OAuthAccessToken

An **OAuthAccessToken** represents an **OAuth** access token, as described in [RFC 6749, section 1.4](#).

An **OAuthAccessToken** is created by a request to the **/oauth/token** endpoint, as described in [RFC 6749, section 4.1.3](#).

Access tokens are used as bearer tokens to authenticate to the API.

OAuthAccessToken Object Definition

```
kind: "OAuthAccessToken"
apiVersion: "v1"
metadata:
  name: "ODliOGE5ZmMtYzcyYi00Nzk1LTg4MGEtNzQyZmUxZmUwY2Vh" 1
  resourceVersion: "1"
  creationTimestamp: "2015-01-01T01:01:02-00:00"
clientName: "openshift-web-console" 2
expiresIn: 86400 3
scopes: []
redirectURI: "https://localhost:8443/console/oauth" 4
```

```

userName: "bob" 5
userID: "9311ac33-0fde-11e5-97a1-3c970e4b7ffe" 6
authorizeToken: "MDAwYjM5YjMtMzM1MC00NDY4LTkxODItOTA2OTE2YzE0M2Fj" 7

```

- 1 **name** is the token name, which is used as a bearer token to authenticate to the API.
- 2 The **clientName** value is the OAuthClient that requested this token.
- 3 The **expiresIn** value is the expiration in seconds from the creationTimestamp.
- 4 The **redirectURI** is where the user was redirected to during the authorization flow that resulted in this token.
- 5 **userName** represents the User this token allows authentication as.
- 6 **userID** represents the User this token allows authentication as.
- 7 **authorizeToken** is the name of the OAuthAuthorizationToken used to obtain this token, if any.

4.6.8. User Objects

4.6.8.1. Identity

When a user logs into OpenShift Dedicated, they do so using a configured identity provider. This determines the user's identity, and provides that information to OpenShift Dedicated.

OpenShift Dedicated then looks for a **UserIdentityMapping** for that **Identity**:

- If the **Identity** already exists, but is not mapped to a **User**, login fails.
- If the **Identity** already exists, and is mapped to a **User**, the user is given an **OAuthAccessToken** for the mapped **User**.
- If the **Identity** does not exist, an **Identity**, **User**, and **UserIdentityMapping** are created, and the user is given an **OAuthAccessToken** for the mapped **User**.

Identity Object Definition

```

kind: "Identity"
apiVersion: "v1"
metadata:
  name: "anypassword:bob" 1
  uid: "9316ebad-0fde-11e5-97a1-3c970e4b7ffe"
  resourceVersion: "1"
  creationTimestamp: "2015-01-01T01:01:01-00:00"
providerName: "anypassword" 2
providerUserName: "bob" 3
user:
  name: "bob" 4
  uid: "9311ac33-0fde-11e5-97a1-3c970e4b7ffe" 5

```

- 1 The identity name must be in the form providerName:providerUserName.

- 2 **providerName** is the name of the identity provider.
- 3 **providerUserName** is the name that uniquely represents this identity in the scope of the identity provider.
- 4 The **name** in the **user** parameter is the name of the user this identity maps to.
- 5 The **uid** represents the UID of the user this identity maps to.

4.6.8.2. User

A **User** represents an actor in the system. Users are granted permissions by adding roles to users or to their groups.

User objects are created automatically on first login, or can be created via the API.



NOTE

OpenShift Dedicated user names containing /, :, and % are not supported.

User Object Definition

```
kind: "User"
apiVersion: "v1"
metadata:
  name: "bob" 1
  uid: "9311ac33-0fde-11e5-97a1-3c970e4b7ffe"
  resourceVersion: "1"
  creationTimestamp: "2015-01-01T01:01:01-00:00"
identities:
  - "anypassword:bob" 2
fullName: "Bob User" 3
```

- 1 **name** is the user name used when adding roles to a user.
- 2 The values in **identities** are Identity objects that map to this user. May be **null** or empty for users that cannot log in.
- 3 The **fullName** value is an optional display name of user.

4.6.8.3. UserIdentityMapping

A **UserIdentityMapping** maps an **Identity** to a **User**.

Creating, updating, or deleting a **UserIdentityMapping** modifies the corresponding fields in the **Identity** and **User** objects.

An **Identity** can only map to a single **User**, so logging in as a particular identity unambiguously determines the **User**.

A **User** can have multiple identities mapped to it. This allows multiple login methods to identify the same **User**.

UserIdentityMapping Object Definition

```
kind: "UserIdentityMapping"
apiVersion: "v1"
metadata:
  name: "anypassword:bob" ❶
  uid: "9316ebad-0fde-11e5-97a1-3c970e4b7ffe"
  resourceVersion: "1"
identity:
  name: "anypassword:bob"
  uid: "9316ebad-0fde-11e5-97a1-3c970e4b7ffe"
user:
  name: "bob"
  uid: "9311ac33-0fde-11e5-97a1-3c970e4b7ffe"
```

- ❶ **UserIdentityMapping** name matches the mapped **Identity** name

4.6.8.4. Group

A **Group** represents a list of users in the system. Groups are granted permissions by adding roles to users or to their groups.

Group Object Definition

```
kind: "Group"
apiVersion: "v1"
metadata:
  name: "developers" ❶
  creationTimestamp: "2015-01-01T01:01:01-00:00"
users:
  - "bob" ❷
```

- ❶ **name** is the group name used when adding roles to a group.
- ❷ The values in **users** are the names of User objects that are members of this group.

CHAPTER 5. NETWORKING

5.1. NETWORKING

5.1.1. Overview

Kubernetes ensures that pods are able to network with each other, and allocates each pod an IP address from an internal network. This ensures all containers within the pod behave as if they were on the same host. Giving each pod its own IP address means that pods can be treated like physical hosts or virtual machines in terms of port allocation, networking, naming, service discovery, load balancing, application configuration, and migration.

Creating links between pods is unnecessary, and it is not recommended that your pods talk to one another directly using the IP address. Instead, it is recommended that you create a [service](#), then interact with the service.

5.1.2. OpenShift Dedicated DNS

If you are running multiple [services](#), such as frontend and backend services for use with multiple pods, in order for the frontend pods to communicate with the backend services, environment variables are created for user names, service IPs, and more. If the service is deleted and recreated, a new IP address can be assigned to the service, and requires the frontend pods to be recreated in order to pick up the updated values for the service IP environment variable. Additionally, the backend service has to be created before any of the frontend pods to ensure that the service IP is generated properly, and that it can be provided to the frontend pods as an environment variable.

For this reason, OpenShift Dedicated has a built-in DNS so that the services can be reached by the service DNS as well as the service IP/port. OpenShift Dedicated supports split DNS by running [SkyDNS](#) on the master that answers DNS queries for services. The master listens to port 53 by default.

When the node starts, the following message indicates the Kubelet is correctly resolved to the master:

```
0308 19:51:03.118430    4484 node.go:197] Started Kubelet for node
openshiftdev.local, server at 0.0.0.0:10250
I0308 19:51:03.118459    4484 node.go:199]   Kubelet is setting 10.0.2.15
as a
DNS nameserver for domain "local"
```

If the second message does not appear, the Kubernetes service may not be available.

On a node host, each container's nameserver has the master name added to the front, and the default search domain for the container will be `.<pod_namespace>.cluster.local`. The container will then direct any nameserver queries to the master before any other nameservers on the node, which is the default behavior for Docker-formatted containers. The master will answer queries on the `.cluster.local` domain that have the following form:

Table 5.1. DNS Example Names

Object Type	Example
Default	<code><pod_namespace>.cluster.local</code>

Object Type	Example
Services	<service>.<pod_namespace>.svc.cluster.local
Endpoints	<name>.<namespace>.endpoints.cluster.local

This prevents having to restart frontend pods in order to pick up new services, which would create a new IP for the service. This also removes the need to use environment variables, because pods can use the service DNS. Also, as the DNS does not change, you can reference database services as **db.local** in configuration files. Wildcard lookups are also supported, because any lookups resolve to the service IP, and removes the need to create the backend service before any of the frontend pods, since the service name (and hence DNS) is established upfront.

This DNS structure also covers headless services, where a portal IP is not assigned to the service and the kube-proxy does not load-balance or provide routing for its endpoints. Service DNS can still be used and responds with multiple A records, one for each pod of the service, allowing the client to round-robin between each pod.

5.2. HAPROXY ROUTER PLUG-IN

5.2.1. Overview

A router is one way to get traffic into the cluster. The HAProxy Template Router plug-in is one of the available router plugins.

5.2.2. HAProxy Template Router

The template router has two components:

- a wrapper that watches endpoints and routes and causes a HAProxy reload based on changes.
- a controller that builds the HAProxy configuration file based on routes and endpoints.



NOTE

The [HAProxy router](#) uses version 1.5.18.

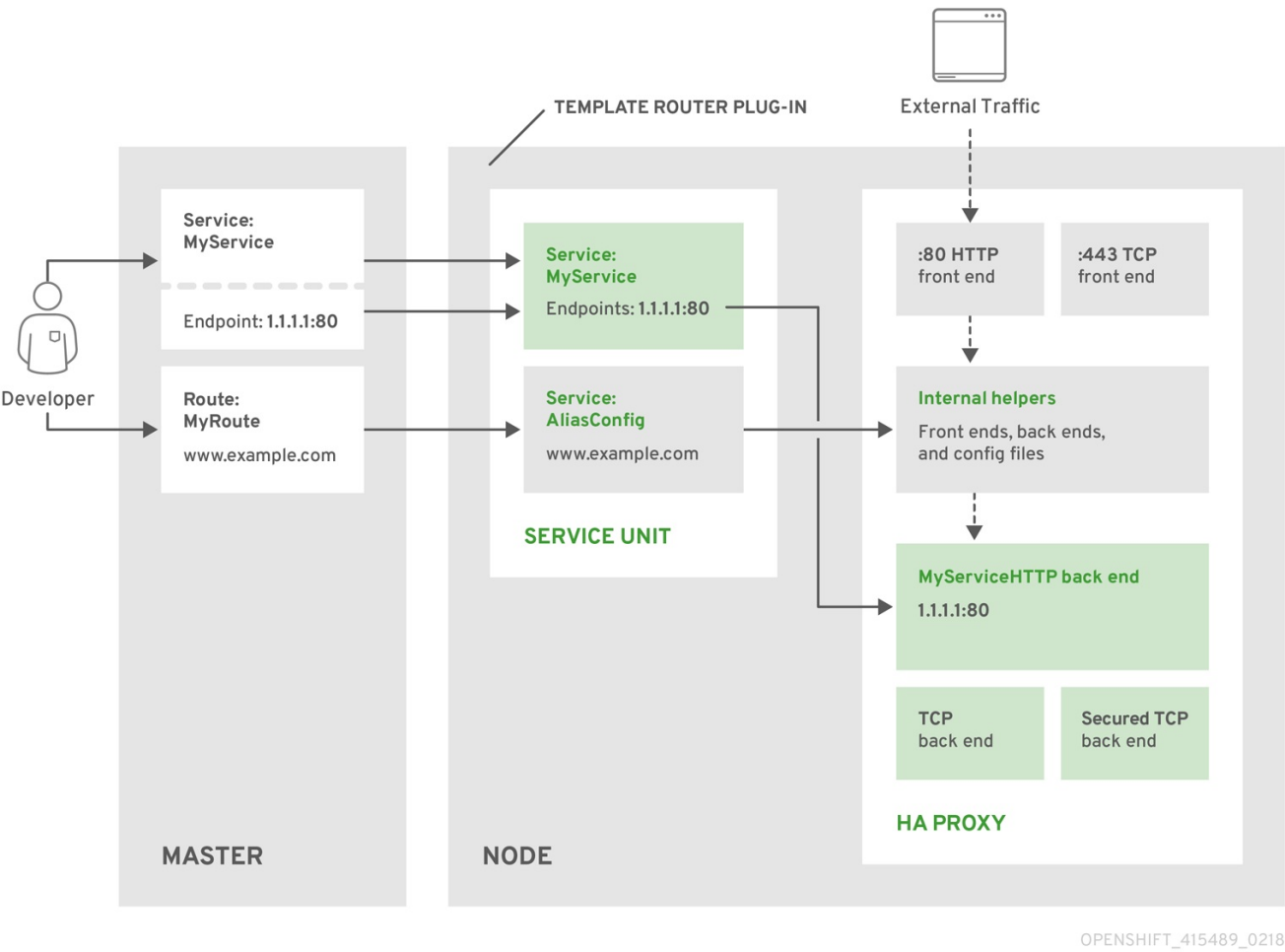
The controller and HAProxy are housed inside a pod, which is managed by a deployment configuration. The process of setting up the router is automated by the **oc adm router** command.

The controller watches the routes and endpoints for changes, as well as HAProxy's health. When a change is detected, it builds a new haproxy-config file and restarts HAProxy. The haproxy-config file is constructed based on the router's template file and information from OpenShift Dedicated.

The HAProxy template file can be customized as needed to support features that are not currently supported by OpenShift Dedicated. The [HAProxy manual](#) describes all of the features supported by HAProxy.

The following diagram illustrates how data flows from the master through the plug-in and finally into an HAProxy configuration:

Figure 5.1. HAProxy Router Data Flow



5.2.2.1. HAProxy Template Router Metrics

The HAProxy router exposes or publishes metrics in [Prometheus format](#) for consumption by external metrics collection and aggregation systems (e.g. Prometheus, statsd). The router can be configured to provide [HAProxy CSV format](#) metrics, or provide no router metrics at all.

The metrics are collected from both the router controller and from HAProxy every 5 seconds. The router metrics counters start at zero when the router is deployed and increase over time. The HAProxy metrics counters are reset to zero every time haproxy is reloaded. The router collects HAProxy statistics for each frontend, backend and server. To reduce resource usage when there are more than 500 servers, the backends are reported instead of the servers since a backend can have multiple servers.

The statistics are a subset of the available HAProxy [Statistics](#).

The following HAProxy metrics are collected on a periodic basis and converted to Prometheus format. For every frontend the "F" counters are collected. When the counters are collected for each backend and the "S" server counters are collected for each server. Otherwise, the "B" counters are collected for each backend and no server counters are collected.

In the following table:

Column 1 - Index from HAProxy CSV statistics

Column 2

F	Frontend metrics
---	------------------

b	Backend metrics when not showing Server metrics due to the Server Threshold,
B	Backend metrics when showing Server metrics
S	Server metrics.

Column 3 - The counter

Column 4 - Counter description

Index	Usage	Counter	Description
2	bBS	current_queue	Current number of queued requests not assigned to any server.
4	FbS	current_sessions	Current number of active sessions.
5	FbS	max_sessions	Maximum observed number of active sessions.
7	FbBS	connections_total	Total number of connections.
8	FbS	bytes_in_total	Current total of incoming bytes.
9	FbS	bytes_out_total	Current total of outgoing bytes.
13	bS	connection_errors_total	Total of connection errors.
14	bS	response_errors_total	Total of response errors.
17	bBS	up	Current health status of the backend (1 = UP, 0 = DOWN).
21	S	check_failures_total	Total number of failed health checks.

24	S	downtime_seconds_total	Total downtime in seconds.", nil),
33	FbS	current_session_rate	Current number of sessions per second over last elapsed second.
35	FbS	max_session_rate	Maximum observed number of sessions per second.
40	FbS	http_responses_total	Total of HTTP responses, code 2xx
43	FbS	http_responses_total	Total of HTTP responses, code 5xx
60	bS	http_average_response_latency_milliseconds	of the last 1024 requests in milliseconds.

The router controller scrapes the following items. These are only available with Prometheus format metrics.

Name	Description
template_router_reload_seconds	Measures the time spent reloading the router in seconds.
template_router_write_config_seconds	Measures the time spent writing out the router configuration to disk in seconds.
haproxy_exporter_up	Was the last scrape of haproxy successful.
haproxy_exporter_csv_parse_failures	Number of errors while parsing CSV.
haproxy_exporter_scrape_interval	The time in seconds before another scrape is allowed, proportional to size of data.
haproxy_exporter_server_threshold	Number of servers tracked and the current threshold value.
haproxy_exporter_total_scrapes	Current total HAProxy scrapes.
http_request_duration_microseconds	The HTTP request latencies in microseconds.
http_request_size_bytes	The HTTP request sizes in bytes.

http_response_size_bytes	The HTTP response sizes in bytes.
openshift_build_info	A metric with a constant '1' value labeled by major, minor, git commit & git version from which OpenShift was built.
ssh_tunnel_open_count	Counter of SSH tunnel total open attempts
ssh_tunnel_open_fail_count	Counter of SSH tunnel failed open attempts

5.3. ROUTES

5.3.1. Overview

An OpenShift Dedicated route exposes a [service](#) at a host name, like *www.example.com*, so that external clients can reach it by name.

DNS resolution for a host name is handled separately from routing. Your administrator may have configured a DNS wildcard entry that will resolve to the OpenShift Dedicated node that is running the OpenShift Dedicated router. If you are using a different host name you may need to modify its DNS records independently to resolve to the node that is running the router.

Each route consists of a name (limited to 63 characters), a service selector, and an optional security configuration.



NOTE

Wildcard routes are disabled in OpenShift Dedicated.

5.3.2. Route-specific Annotations

Using environment variables, a router can set the default options for all the routes it exposes. An individual route can override some of these defaults by providing specific configurations in its annotations.

Route Annotations

For all the items outlined in this section, you can set annotations on the **route definition** for the route to alter its configuration

Table 5.2. Route Annotations

Variable	Description	Environment Variable Used as Default
<code>haproxy.router.openshift.io/balance</code>	Sets the load-balancing algorithm. Available options are source , roundrobin , and leastconn .	ROUTER_TCP_BALANCE_SCHEME for passthrough routes. Otherwise, use ROUTER_LOAD_BALANCE_ALGORITHM .

Variable	Description	Environment Variable Used as Default
<code>haproxy.router.openshift.io/disable_cookies</code>	Disables the use of cookies to track related connections. If set to true or TRUE , the balance algorithm is used to choose which back-end serves connections for each incoming HTTP request.	
<code>haproxy.router.openshift.io/cookie_name</code>	Specifies an optional cookie to be used for this route. The name must consist of any combination of upper and lower case letters, digits, "_", and "-". The default is the hashed internal key name for the route.	
<code>haproxy.router.openshift.io/rate-limit-connections</code>	Setting true or TRUE to enables rate limiting functionality.	
<code>haproxy.router.openshift.io/rate-limit-connections.concurrent-tcp</code>	Limits the number of concurrent TCP connections shared by an IP address.	
<code>haproxy.router.openshift.io/rate-limit-connections.rate-http</code>	Limits the rate at which an IP address can make HTTP requests.	
<code>haproxy.router.openshift.io/rate-limit-connections.rate-tcp</code>	Limits the rate at which an IP address can make TCP connections.	
<code>haproxy.router.openshift.io/timeout</code>	Sets a server-side timeout for the route. (TimeUnits)	ROUTER_DEFAULT_SERVER_TIMEOUT
<code>router.openshift.io/haproxy.health.check.interval</code>	Sets the interval for the back-end health checks. (TimeUnits)	ROUTER_BACKEND_CHECK_INTERVAL
<code>haproxy.router.openshift.io/ip_whitelist</code>	Sets a whitelist for the route.	
<code>haproxy.router.openshift.io/hsts_header</code>	Sets a Strict-Transport-Security header for the edge terminated or re-encrypt route.	

Example 5.1. A Route Setting Custom Timeout

```

apiVersion: v1
kind: Route
metadata:
  annotations:
    haproxy.router.openshift.io/timeout: 5500ms ❶
[...]
```

- ❶ Specifies the new timeout with HAProxy supported units (us, ms, s, m, h, d). If unit not provided, ms is the default.



NOTE

Setting a server-side timeout value for passthrough routes too low can cause WebSocket connections to timeout frequently on that route.

5.3.3. Route-specific IP Whitelists

You can restrict access to a route to a select set of IP addresses by adding the **haproxy.router.openshift.io/ip_whitelist** annotation on the route. The whitelist is a space-separated list of IP addresses and/or CIDRs for the approved source addresses. Requests from IP addresses that are not in the whitelist are dropped.

Some examples:

When editing a route, add the following annotation to define the desired source IP's. Alternatively, use **oc annotate route <name>**.

Allow only one specific IP address:

```

metadata:
  annotations:
    haproxy.router.openshift.io/ip_whitelist: 192.168.1.10
```

Allow several IP addresses:

```

metadata:
  annotations:
    haproxy.router.openshift.io/ip_whitelist: 192.168.1.10 192.168.1.11
192.168.1.12
```

Allow an IP CIDR network:

```

metadata:
  annotations:
    haproxy.router.openshift.io/ip_whitelist: 192.168.1.0/24
```

Allow mixed IP addresses and IP CIDR networks:

```

metadata:
  annotations:
    haproxy.router.openshift.io/ip_whitelist: 180.5.61.153 192.168.1.0/24
```

10.0.0.0/8