OpenShift Container Platform 3.4
Architecture

OpenShift Container Platform 3.4 Architecture Information

Red Hat OpenShift Documentation Team

OpenShift Container Platform 3.4 Architecture

OpenShift Container Platform 3.4 Architecture Information

Legal Notice

Abstract

Learn the architecture of OpenShift Container Platform 3.4 including the infrastructure and core components. These topics also cover authentication, networking and source code management.

Table of Contents

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 Container Platform 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

Figure 1.1. OpenShift Container Platform Architecture Overview
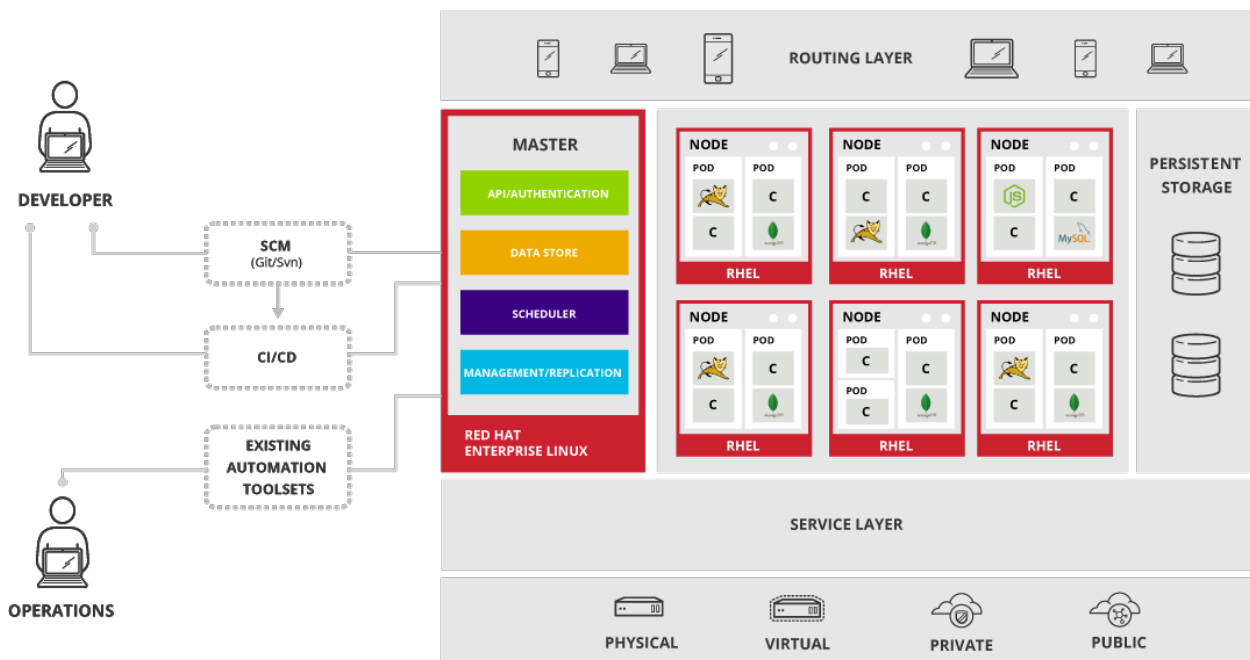
1.2. WHAT IS THE OPENSHIFT CONTAINER PLATFORM ARCHITECTURE?

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

» REST APIs, which expose each of the core objects.

» 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 Container Platform is extensible. The way that builds are run and launched can be customized independently of how images are managed, or how deployments happen. The controllers are performing the "business logic" of the system, taking user actions and transforming them into reality. By customizing those controllers or replacing them with your own logic, different 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 Container Platform 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 OPENSHIFT CONTAINER PLATFORM SECURED?

The OpenShift Container Platform 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 SSL certificate authorization.

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 Container Platform 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.

Since every container that runs on the cluster is associated with a service account, it is also possible to associate secrets to those service accounts and have them automatically delivered into the container. This enables the infrastructure to manage secrets for pulling and pushing images, builds, and the deployment components, and also allows application code to easily leverage those secrets.

CHAPTER 2. INFRASTRUCTURE COMPONENTS

2.1. KUBERNETES INFRASTRUCTURE

2.1.1. Overview

Within OpenShift Container Platform, 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 Container Platform uses Kubernetes 1.4 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.

Component	Description
HAProxy	Optional, used when configuring highly-available masters with the **native** method to balance load between API master endpoints. The advanced installation method can configure HAProxy for you with the **native** method. Alternatively, you can use the **native** method but pre-configure your own load balancer of choice.

2.1.2.1. High Availability Masters

While in a single master configuration, 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. You can optionally configure your masters for high availability (HA) to ensure that the cluster has no single point of failure.

To mitigate concerns about availability of the master, two activities are recommended:

1. A runbook entry should be created for reconstructing the master. A runbook entry is a necessary backstop for any highly-available service. Additional solutions merely control the frequency that the runbook must be consulted. For example, a cold standby of the master host can adequately fulfill SLAs that require no more than minutes of downtime for creation of new applications or recovery of failed application components.

2. Use a high availability solution to configure your masters and ensure that the cluster has no single point of failure. The advanced installation method 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.

> **Note**
>
> Moving from a single master cluster to multiple masters after installation is not supported.

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

Role	Style	Notes
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 Container Platform 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.

Administrators can manage nodes in an OpenShift Container Platform instance using the CLI. To define full configuration and security options when launching node servers, use dedicated node configuration files.

> **Important**
>
> The recommended maximum number of nodes is 300.

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 sample manifest can be found in the Kubernetes documentation.

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  1
kind: Node  2
metadata:
  creationTimestamp: null
  labels:  3
    kubernetes.io/hostname: node1.example.com
  name: node1.example.com  4
spec:
  externalID: node1.example.com  5
status:
  nodeInfo:
    bootID: ""
    containerRuntimeVersion: ""
    kernelVersion: ""
    kubeProxyVersion: ""
    kubeletVersion: ""
    machineID: ""
    osImage: ""
    systemUUID: ""
```

1

apiVersion defines the API version to use.

2

kind set to **Node** identifies this as a definition for a node object.

3

metadata.labels lists any labels that have been added to the node.

4

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.

5

> **spec.externalID** defines the fully-qualified domain name where the node can be reached. Defaults to the **metadata.name** value when empty.

The REST API Reference has more details on these definitions.

2.2. CONTAINER REGISTRY

2.2.1. Overview

OpenShift Container Platform 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 Container Platform registry.

2.2.2. Integrated OpenShift Container Registry

OpenShift Container Platform 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 Container Platform about the new image, passing along all the information about it, such as the namespace, name, and image metadata. Different pieces of OpenShift Container Platform react to new images, creating new builds and deployments.

OCR can also be deployed as a stand-alone component that acts solely as a container registry, without the build and deployment integration. See Installing a Stand-alone Deployment of OpenShift Container Registry for details.

2.2.3. Third Party Registries

OpenShift Container Platform 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 Container Platform registry. In this situation OpenShift Container Platform 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 Container Platform 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 Container Platform 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.

The web console is started as part of the master. All static assets required to run the web console are served from the **openshift** binary. Administrators can also customize the web console using extensions, which let you run scripts and load custom stylesheets when the web console loads. You can change the look and feel of nearly any aspect of the user interface in this way.

When you access the web console from a browser, it first loads all required static assets. It then makes requests to the OpenShift Container Platform APIs using the values defined from the **openshift start** option **--public-master**, or from the related master configuration file parameter **masterPublicURL**. The web console uses WebSockets to maintain a persistent connection with the API server and receive updated information as soon as it is available.

Figure 2.1. Web Console Request Architecture
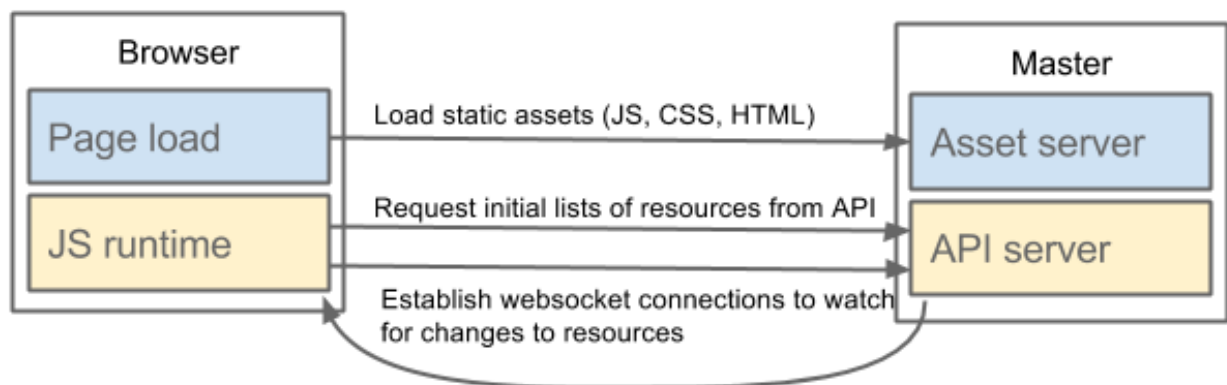
The configured host names and IP addresses for the web console are whitelisted to access the API server safely even when the browser would consider the requests to be cross-origin. To access the API server from a web application using a different host name, you must whitelist that host name by specifying the **--cors-allowed-origins** option on **openshift start** or from the related master configuration file parameter **corsAllowedOrigins**.

2.3.2. CLI Downloads

You can download and unpack the CLI from the **Command Line Tools** page on the web console for use on Linux, MacOSX, and Windows clients. Cluster administrators can customize these links further.

2.3.3. Browser Requirements

Review the tested integrations for OpenShift Container Platform. The following browser versions and operating systems can be used to access the web console.

Table 2.3. Browser Requirements

Browser (Latest Stable)	Operating System
Firefox	Fedora 23, Windows 8
Internet Explorer	Windows 8
Chrome	Fedora 23, Windows 8, and MacOSX
Safari	MacOSX, iPad 2, iPhone 4

2.3.4. CLI Downloads

You can download and unpack the CLI from the **About** page on the web console for use on Linux, MacOSX, and Windows clients if your cluster administrator has enabled it :
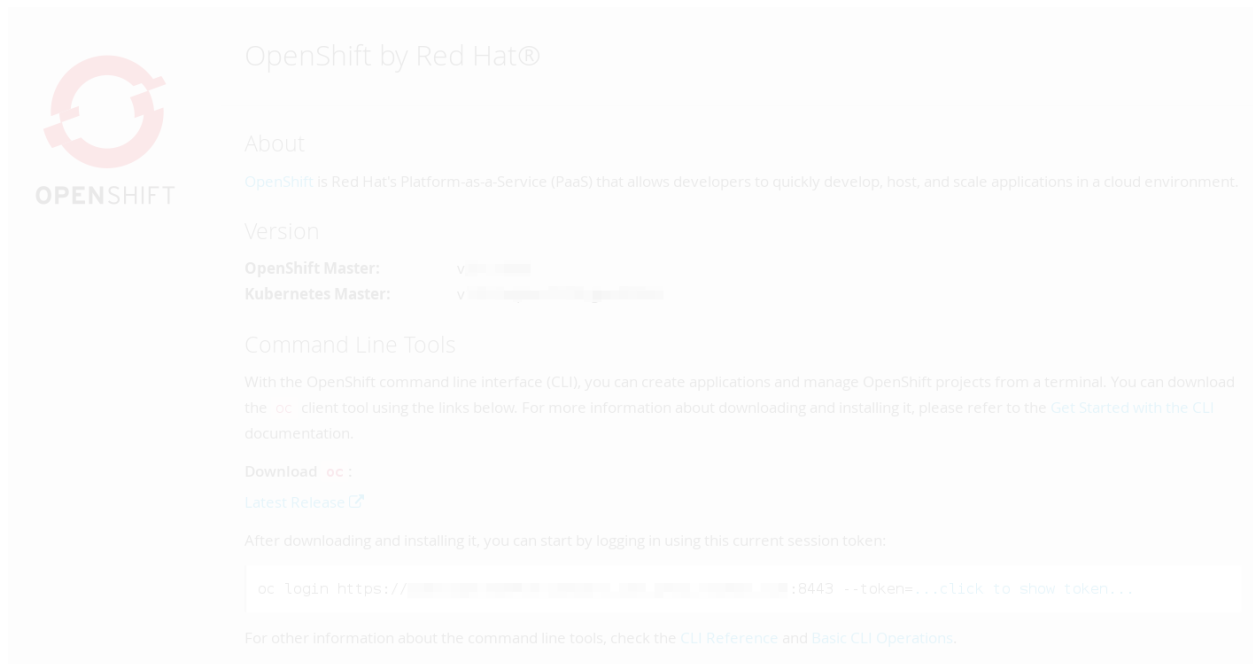
2.3.5. Project Overviews

After logging in, the web console provides developers with an overview for the currently selected project:

Figure 2.2. Web Console Project Overview
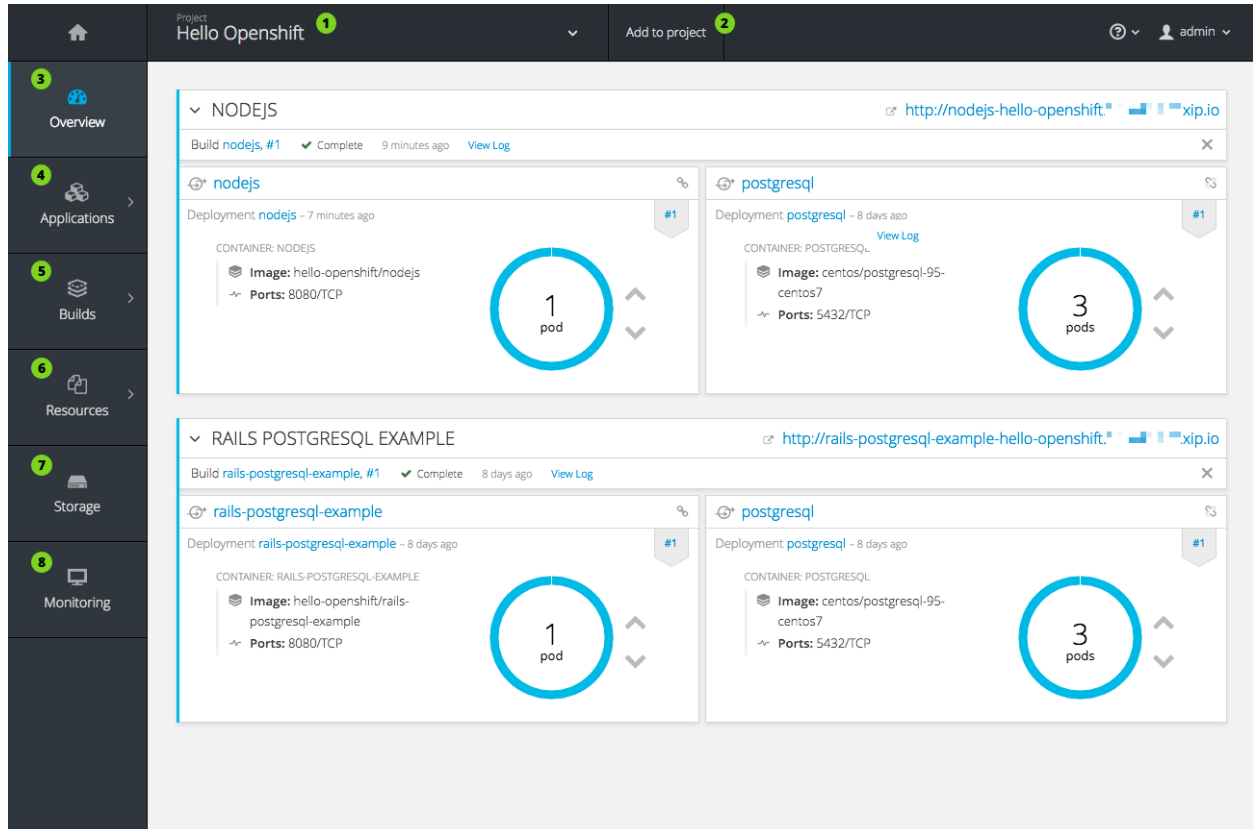
The project selector allows you to switch between projects you have access to.

Create new applications using a source repository or using a template.

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.

> **Note**
>
> Cockpit is automatically installed and enabled in OpenShift Container Platform 3.1 and later to help you monitor your development environment. Red Hat Enterprise Linux Atomic Host: Getting Started with Cockpit provides more information on using Cockpit.

2.3.6. 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 2.3. 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 2.4. 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.

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 Container Platform. Many of these objects come from Kubernetes, which is extended by OpenShift Container Platform 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 Container Platform 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 Container Platform 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 Container Platform, understanding their capabilities and terminology is important for understanding their role in OpenShift Container Platform 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 Container Platform. Refer to the article Get Started with Docker Formatted Container Images on Red Hat Systems for a guided introduction.

3.2.1.1. Init Containers

A pod can have init containers in addition to application containers. Init containers allow you to reorganize setup scripts and binding code. An init container differs from a regular container in that it always runs to completion. Each init container must complete successfully before the next one is started.

The Kubernetes documentation has more information on init containers, including usage examples.

3.2.2. Images

Containers in OpenShift Container Platform 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 Container Platform 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 Container Platform 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**.

Do not use the **latest** tag for any official OpenShift Container Platform 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 Container Platform 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 Container Platform 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:
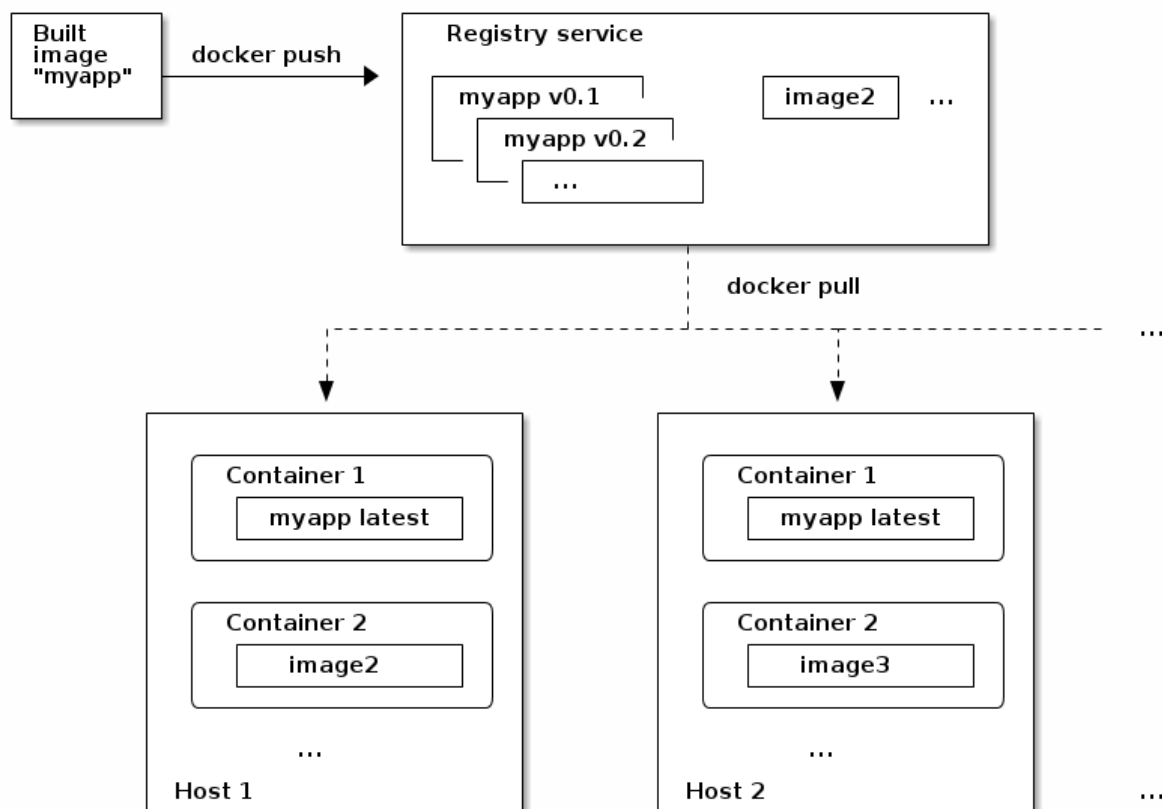
3.3. PODS AND SERVICES

3.3.1. Pods

OpenShift Container Platform 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 Container Platform treats pods as largely immutable; changes cannot be made to a pod definition while it is running. OpenShift Container Platform 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 Container Platform node host is 110.

Below is an example definition of a pod that provides a long-running service, which is actually a part of the OpenShift Container Platform 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:                                        1
    deployment: docker-registry-1
    deploymentconfig: docker-registry
    docker-registry: default
  generateName: docker-registry-1-              2
spec:
  containers:                                    3
  - env:                                         4
    - name: OPENSHIFT_CA_DATA
      value: ...
    - name: OPENSHIFT_CERT_DATA
      value: ...
    - name: OPENSHIFT_INSECURE
      value: "false"
    - name: OPENSHIFT_KEY_DATA
      value: ...
    - name: OPENSHIFT_MASTER
      value: https://master.example.com:8443
    image: openshift/origin-docker-registry:v0.6.2   5
    imagePullPolicy: IfNotPresent
    name: registry
```

```
      ports:                                    6
      - containerPort: 5000
        protocol: TCP
      resources: {}
      securityContext: { ... }                  7
      volumeMounts:                             8
      - 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                       9

  volumes:                                      10
  - emptyDir: {}
    name: registry-storage
  - name: default-token-br6yz
    secret:
      secretName: default-token-br6yz
```

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 Container Platform 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 Container Platform API.

9

Pods making requests against the OpenShift Container Platform 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.

10

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 Container Platform automatically after the pod is created and its lifecycle begins. The Kubernetes API documentation has complete details of the pod REST API object attributes, and the Kubernetes pod documentation has details about the functionality and purpose of pods.

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 Container

Platform internal network and they are used to permit pods to access each other. To permit external access to the service, additional externalIP addresses that are external to cluster, can be assigned to the service. These externalIP addresses can also be virtual IP addresses that provide highly available access to the service.

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        1
spec:
  selector:                    2
    docker-registry: default
  portalIP: 172.30.136.123     3
  ports:
  - nodePort: 0
    port: 5000                 4
    protocol: TCP
    targetPort: 5000           5
```

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 externalIPs

In addition to the cluster's internal IP addresses, the user can configure IP addresses that are external to the cluster. The administrator is responsible for ensuring that traffic arrives at a node with this IP.

The externalIPs must be selected by the admin from the **ExternalIPNetworkCIDRs** range configured in *master-config.yaml* file. When *master-config.yaml* is changed, the master service must be restarted.

Example 3.3. Sample ExternalIPNetworkCIDR /etc/origin/master/master-config.yaml

```
networkConfig:
  ExternalIPNetworkCIDR: 172.47.0.0/24
```

Example 3.4. Service externalIPs Definition (JSON)

```
{
    "kind": "Service",
    "apiVersion": "v1",
    "metadata": {
        "name": "my-service"
    },
    "spec": {
        "selector": {
            "app": "MyApp"
        },
        "ports": [
            {
                "name": "http",
                "protocol": "TCP",
                "port": 80,
                "targetPort": 9376
            }
        ],
        "externalIPs" : [
            "80.11.12.10"            1
        ]
    }
}
```

1

List of External IP addresses on which the **port** is exposed. In addition to the internal IP addresses)

3.3.2.2. Service ingressIPs

In non-cloud clusters, externalIP addresses can be automatically assigned from a pool of addresses. This eliminates the need for the administrator manually assigning them.

The pool is configured in ***/etc/origin/master/master-config.yaml*** file. After changing this file, restart the master service.

The `ingressIPNetworkCIDR` is set to **172.29.0.0/16** by default. If the cluster environment is not already using this private range, use the default range or set a custom range.

> **Note**
>
> If you are using high availibility, then this range must be less than 256 addresses.

Example 3.5. Sample ingressIPNetworkCIDR /etc/origin/master/master-config.yaml

```
networkConfig:
  ingressIPNetworkCIDR: 172.29.0.0/16
```

3.3.2.3. Service NodePort

Setting the service **type=NodePort** will allocate a port from a flag-configured range (default: 30000-32767), and each node will proxy that port (the same port number on every node) into your service.

The selected port will be reported in the service configuration, under `spec.ports[*].nodePort`.

To specify a custom port just place the port number in the nodePort field. The custom port number must be in the configured range for nodePorts. When '**master-config.yaml**' is changed the master service must be restarted.

Example 3.6. Sample servicesNodePortRange /etc/origin/master/master-config.yaml

```
kubernetesMasterConfig:
  servicesNodePortRange: ""
```

The service will be visible as both the **`<NodeIP>:spec.ports[].nodePort`** and **`spec.clusterIp:spec.ports[].port`**

> **Note**
>
> Setting a nodePort is a privileged operation.

3.3.2.4. Service Proxy Mode

OpenShift Container Platform has two different implementations of the service-routing infrastructure. The default implementation is entirely **iptables**-based, and uses probabilistic **iptables** rewriting rules to distribute incoming service connections between the endpoint pods. The older implementation uses a user space process to accept incoming connections and then proxy traffic between the client and one of the endpoint pods.

The **iptables**-based implementation is much more efficient, but it requires that all endpoints are always able to accept connections; the user space implementation is slower, but can try multiple endpoints in turn until it finds one that works. If you have good readiness checks (or generally reliable nodes and pods), then the **iptables**-based service proxy is the best choice. Otherwise, you can enable the user space-based proxy when installing, or after deploying the cluster by editing the node configuration file.

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 Container Platform 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 Container Platform cluster. In this case, you can leave out the **selector** field in the service, and create the Endpoints object manually.

Note that OpenShift Container Platform 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 Container Platform is associated with a user. An OpenShift Container Platform 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 Container Platform 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 Container Platform. 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 Container Platform, 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.

3.5. 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 Container Platform 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 Container Platform 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 Container Platform 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.	
Reproducib ility	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 Container Platform 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 Container Platform instantiates a Jenkins server to execute the pipeline. Subsequent Pipeline build configurations in the project share this Jenkins server.

For more details on how the Jenkins server is deployed and how to configure or disable the autoprovisioning behavior, see Configuring Pipeline Execution.

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, please see the Jenkins documentation.

3.5.2. Image Streams

An *image stream* comprises any number of Docker-formatted container images identified by tags. It presents a single virtual view of related images, similar to an image repository, and may contain images from any of the following:

1. Its own image repository in OpenShift Container Platform's integrated registry

2. Other image streams

3. Image repositories from external registries

Image streams can be used to automatically perform an action when new images are created. Builds and deployments can watch an image stream to receive notifications when new images are added and react by performing a build or deployment, respectively.

> **Note**
>
> See the Developer Guide for details on managing images and image streams.

Example 3.7. Image Stream Object Definition

```
apiVersion: v1
kind: ImageStream
metadata:
  annotations:
    openshift.io/generated-by: OpenShiftNewApp
  creationTimestamp: 2016-01-29T13:33:49Z
  generation: 1
  labels:
    app: ruby-sample-build
    template: application-template-stibuild
  name: origin-ruby-sample
  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
  tags:
  - items:
    - created: 2016-01-29T13:40:11Z
      dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135b
f7dd13d
      generation: 1
      image:
sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d
    tag: latest
```

3.5.2.1. Image Stream Image

An *image stream image* is a virtual resource that allows you to retrieve an image from a particular *image stream* where it is tagged. It is often abbreviated as *isimage*. It consists of two parts delimited by an at sign: **<image stream name>@<image name>**. To refer to the image in the example above, the *isimage* looks like:

```
origin-ruby-
sample@sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135b
f7dd13d
```

Users, without permission to read or list images on the cluster level, can still retrieve the images tagged in a project they have access to using this resource.

3.5.2.2. Image Stream Tag

An *image stream tag* is a named pointer to an image in an *image stream*. It is often abbreviated as *istag*. It 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 *istag*, it is just placed at the first position in the history stack. Image previously occupying the top position will be available at the second position etc. This allows for easy rollbacks to make tags point to historical images again.

The *istag* is composed of two parts separated by a colon: **<image stream name>:<tag>**. *Istag* referring to the image **sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d** in the example above would be **origin-ruby-sample:latest**.

Example 3.8. Image Stream Tag with Two Images in its History

```
  tags:
  - items:
    - created: 2016-03-02T10:15:09Z
      dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:909de62d1f609a717ec433cc25ca5cf00941545c83a01fb31527771e1
fab3fc5
      generation: 2
      image:
sha256:909de62d1f609a717ec433cc25ca5cf00941545c83a01fb31527771e1fab3fc5
    - created: 2016-01-29T13:40:11Z
      dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135b
f7dd13d
      generation: 1
      image:
sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d
    tag: latest
```

3.5.2.3. Image Stream Mappings

When the integrated registry receives a new image, it creates and sends an **ImageStreamMapping** to OpenShift Container Platform, providing the image's namespace (i.e., its project), name, tag, and image metadata.

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 Container Platform stores complete metadata about each image, such as commands, entrypoint, and environment variables. Images in OpenShift Container Platform are immutable and the maximum name length is 63 characters.

> **Note**
>
> See the Developer Guide for details on manually tagging images.

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

Example 3.9. 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-entrypoint
      Env:
```

```
      - RACK_ENV=production
      - OPENSHIFT_BUILD_NAMESPACE=test
      - OPENSHIFT_BUILD_SOURCE=https://github.com/openshift/ruby-hello-
world.git
      - EXAMPLE=sample-app
      - OPENSHIFT_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:
00cadc392d39d5ef9117cbc8a31db0889eedd442
      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@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e
8986b28e
      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.expose-services: 8080:http
      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-entrypoint
        Env:
        - RACK_ENV=production
        - OPENSHIFT_BUILD_NAME=ruby-sample-build-1
        - OPENSHIFT_BUILD_NAMESPACE=test
        - OPENSHIFT_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@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e
8986b28e
        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@sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135b
f7dd13d
```

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      1
  selector:        2
    name: frontend
  template:        3
    metadata:
      labels:      4
        name: frontend  5
    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. 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.

3.6.3. Deployments and Deployment Configurations

Building on replication controllers, OpenShift Container Platform 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 Container Platform 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 Container Platform **DeploymentConfiguration** 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 **DeploymentConfiguration** 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. ROUTES

3.7.1. Overview

An OpenShift Container Platform 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 Container Platform node that is running the OpenShift Container Platform 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.

3.7.2. Routers

An OpenShift Container Platform administrator can deploy *routers* to nodes in an OpenShift Container Platform cluster, which enable routes created by developers to be used by external clients. The routing layer in OpenShift Container Platform is pluggable, and two available router plug-ins are provided and supported by default.

> **Note**
>
> See the Installation and Configuration guide for information on deploying a router.

A router uses the service selector to find the service and the endpoints backing the service. When both router and service provide load balancing, OpenShift Container Platform uses the router load balancing. A router detects relevant changes in the IP addresses of its services and adapts its configuration accordingly. This is useful for custom routers to communicate modifications of API objects to an external routing solution.

The path of a request starts with the DNS resolution of a host name to one or more routers. The suggested method is to define a cloud domain with a wildcard DNS entry pointing to one or more virtual IP (VIP) addresses backed by multiple router instances. Routes using names and addresses outside the cloud domain require configuration of individual DNS entries.

When there are fewer VIP addresses than routers, the routers corresponding to the number of addresses are *active* and the rest are *passive*. A passive router is also known as a *hot-standby* router. For example, with two VIP addresses and three routers, you have an "active-active-passive" configuration. See High Availability for more information on router VIP configuration.

Routes can be sharded among the set of routers. Administrators can set up sharding on a cluster-wide basis and users can set up sharding for the namespace in their project. Sharding allows the operator to define multiple router groups. Each router in the group serves only a subset of traffic.

OpenShift Container Platform routers provide external host name mapping and load balancing of service end points over protocols that pass distinguishing information directly to the router; the host name must be present in the protocol in order for the router to determine where to send it.

Router plug-ins assume they can bind to host ports 80 (HTTP) and 443 (HTTPS), by default. This means that routers must be placed on nodes where those ports are not otherwise in use. Alternatively, a router can be configured to listen on other ports by setting the **ROUTER_SERVICE_HTTP_PORT** and **ROUTER_SERVICE_HTTPS_PORT** environment variables.

Because a router binds to ports on the host node, only one router listening on those ports can be on each node if the router uses host networking (the default). Cluster networking is configured such that all routers can access all pods in the cluster.

Routers support the following protocols:

* HTTP

* HTTPS (with SNI)

* WebSockets

≫ TLS with SNI

> **Note**
>
> WebSocket traffic uses the same route conventions and supports the same TLS termination types as other traffic.

3.7.2.1. Template Routers

A *template router* is a type of router that provides certain infrastructure information to the underlying router implementation, such as:

≫ A wrapper that watches endpoints and routes.

≫ Endpoint and route data, which is saved into a consumable form.

≫ Passing the internal state to a configurable template and executing the template.

≫ Calling a reload script.

3.7.3. Available Router Plug-ins

The following router plug-ins are provided and supported in OpenShift Container Platform. Instructions on deploying these routers are available in Deploying a Router.

3.7.3.1. HAProxy Template Router

The HAProxy template router implementation is the reference implementation for a template router plug-in. It uses the **openshift3/ose-haproxy-router** repository to run an HAProxy instance alongside the template router plug-in.

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

Figure 3.1. HAProxy Router Data Flow
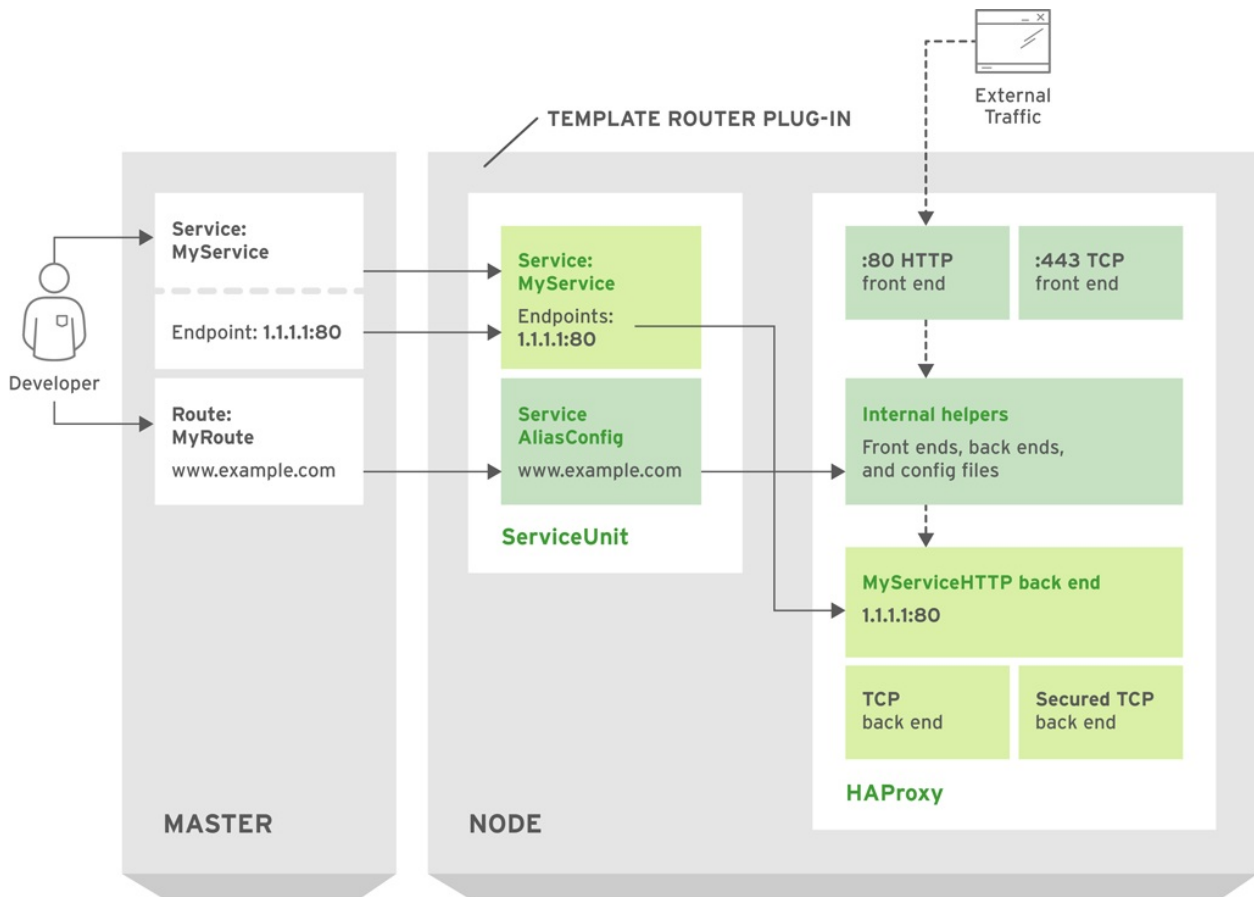
OPENSHIFT3_357398_0615

Sticky Sessions

Sticky sessions attempts to ensure that user traffic is directed to the same pod in each session, and improves application user experience by caching data. For example, for a cluster with five back-end pods, and two load-balanced routers, you can ensure that the same pod receives the web traffic from the same web browser regardless of the router that handles it.

While placing returning traffic onto the same pod cannot be guaranteed, you can use HTTP headers to set a cookie determining the pod used by the last connection, in an attempt to steer traffic to the same pod. When the application user hits the router the browser re-sends the cookie and knows where to send the traffic.

As a cluster administrator, you can turn off the stickiness for passthrough routes separately from other connections, or turn off stickiness entirely.

By default, if the default HAProxy router is using passthrough routes, sticky sessions are implemented using the **balance source** directive in the underlying router configuration, which balances based on the source IP. Other types of routes use **balance leastconn** by default. If a cookie is present, it is read and sent to the same pod it went to last time if the pod still exists. If no cookie is present, because this is a new session, stickiness is disabled, or it's using a passthrough route, the router picks a pod.

> **Note**
>
> Cookies cannot be set on passthrough routes, because the HTTP traffic cannot be seen. Instead, a number is calculated based on the source IP address, which determines the back-end.
>
> If back-ends change, the traffic could head to the wrong server, making it less sticky, and if you are using a load-balancer (which hides the source IP) the same number is set for all connections and traffic is sent to the same pod.

In addition, the template router plug-in provides the service name and namespace to the underlying implementation. This can be used for more advanced configuration such as implementing stick-tables that synchronize between a set of peers.

Specific configuration for this router implementation is stored in the ***haproxy-config.template*** file located in the ***/var/lib/haproxy/conf*** directory of the router container.

> **Note**
>
> The **balance source** directive does not distinguish between external client IP addresses; because of the NAT configuration, the originating IP address (HAProxy remote) is the same. Unless the HAProxy router is running with **hostNetwork: true**, all external clients will be routed to a single pod.

3.7.4. Configuration Parameters

For all the items outlined in this section, you can set environment variables on the **deployment config** for the router to alter its configuration, or use the **oc set env** command:

```
$ oc set env <object_type>/<object_name> KEY1=VALUE1 KEY2=VALUE2
```

For example:

```
$ oc set env dc/router ROUTER_SYSLOG_ADDRESS=127.0.0.1
ROUTER_LOG_LEVEL=debug
```

Table 3.1. Router Configuration Parameters

Variable	Default	Description
DEFAULT_CERT IFICATE		The contents of a default certificate to use for routes that don't expose a TLS server cert; in PEM format.

Variable	Default	Description
DEFAULT_CERT IFICATE_DIR		A path to a directory that contains a file named *tls.crt*. If *tls.crt* is not a PEM file which also contains a private key, it is first combined with a file named tls.key in the same directory. The PEM-format contents are then used as the default certificate. Only used if **DEFAULT_CERTIFICATE** or **DEFAULT_CERTIFICATE_PATH** are not specified.
DEFAULT_CERT IFICATE_PATH		A path to default certificate to use for routes that don't expose a TLS server cert; in PEM format. Only used if **DEFAULT_CERTIFICATE** is not specified.
EXTENDED_VAL IDATION	**true**	If **true**, perform an additional extended validation step on all routes admitted by this router.
NAMESPACE_LA BELS		A label selector to apply to namespaces to watch, empty means all.
PROJECT_LABE LS		A label selector to apply to projects to watch, emtpy means all.
RELOAD_SCRIP T		The path to the reload script to use to reload the router.
ROUTER_ALLOW ED_DOMAINS		A comma-separated list of domains that the host name in a route can only be part of. Any subdomain in the domain can be used. Option **ROUTER_DENIED_DOMAINS** overrides any values given in this option. If set, everything outside of the allowed domains will be rejected.
ROUTER_BACKE ND_CHECK_INT ERVAL	5000ms	Length of time between subsequent "liveness" checks on backends. (TimeUnits)

Variable	Default	Description
ROUTER_COMPRESSION_MIME	"text/html text/plain text/css"	A space separated list of mime types to compress.
ROUTER_DEFAULT_CLIENT_TIMEOUT	30s	Length of time within which a client has to acknowledge or send data. (TimeUnits)
ROUTER_DEFAULT_CONNECT_TIMEOUT	5s	The maximum connect time. (TimeUnits)
ROUTER_DEFAULT_SERVER_TIMEOUT	30s	Length of time within which a server has to acknowledge or send data. (TimeUnits)
ROUTER_DEFAULT_TUNNEL_TIMEOUT	1h	Length of time till which TCP or WebSocket connections will remain open. If you have websockets/tcp connections (and any time HAProxy is reloaded), the old HAProxy processes will "linger" around for that period. (TimeUnits)
ROUTER_DENIED_DOMAINS		A comma-separated list of domains that the host name in a route can not be part of. No subdomain in the domain can be used either. Overrides option **ROUTER_ALLOWED_DOMAINS**.
ROUTER_ENABLE_COMPRESSION		If **true** or **TRUE**, compress responses when possible.
ROUTER_LOG_LEVEL	warning	The log level to send to the syslog server.
ROUTER_MAX_CONNECTIONS	20000	Maximum number of concurrent connections.

Variable	Default	Description
ROUTER_OVERR IDE_HOSTNAME		If set **true**, override the spec.host value for a route with the template in **ROUTER_SUBDOMAIN**.
ROUTER_SERVI CE_HTTPS_POR T	443	Port to listen for HTTPS requests.
ROUTER_SERVI CE_HTTP_PORT	80	Port to listen for HTTP requests.
ROUTER_SERVI CE_NAME	public	The name that the router will identify itself with in route statuses.
ROUTER_SERVI CE_NAMESPACE		The namespace the router will identify itself with in route statuses. Required if **ROUTER_SERVICE_NAME** is used.
ROUTER_SERVI CE_NO_SNI_PO RT	10443	Internal port for some front-end to back-end communication (see note below).
ROUTER_SERVI CE_SNI_PORT	10444	Internal port for some front-end to back-end communication (see note below).
ROUTER_SLOWL ORIS_HTTP_KE EPALIVE	300s	Set the maximum time to wait for a new HTTP request to appear. If this is set too low, it can confuse browsers and applications not expecting a small **keepalive** value. (TimeUnits)
ROUTER_SLOWL ORIS_TIMEOUT	10s	Length of time the transmission of an HTTP request can take. (TimeUnits)
ROUTER_SUBDO MAIN		The template that should be used to generate the host name for a route without spec.host (e.g. ${name}-${namespace}.myapps.mycompany.com).

Variable	Default	Description
ROUTER_SYSLOG_ADDRESS		Address to send log messages. Disabled if empty.
ROUTER_TCP_BALANCE_SCHEME	source	Load-balancing strategy for multiple endpoints for pass-through routes. Available options are **source**, **roundrobin**, or **leastconn**.
ROUTER_LOAD_BALANCE_ALGORITHM	leastconn	Load-balancing strategy routes with multiple endpoints. Available options are **source**, **roundrobin**, and **leastconn**.
ROUTE_LABELS		A label selector to apply to the routes to watch, empty means all.
STATS_PASSWORD		The password needed to access router stats (if the router implementation supports it).
STATS_PORT		Port to expose statistics on (if the router implementation supports it). If not set, stats are not exposed.
STATS_USERNAME		The user name needed to access router stats (if the router implementation supports it).
TEMPLATE_FILE	**/var/lib/haproxy/conf/custom/ haproxy-config-custom.template**	The path to the HAProxy template file (in the container image).
RELOAD_INTERVAL	12s	The minimum frequency the router is allowed to reload to accept new changes. (TimeUnits)

Variable	Default	Description
`ROUTER_USE_PROXY_PROTOCOL`		When set to **true** or **TRUE**, HAProxy expects incoming connections to use the **PROXY** protocol on port 80 or port 443. The source IP address can pass through a load balancer if the load balancer supports the protocol, for example Amazon ELB.
`ROUTER_ALLOW_WILDCARD_ROUTES`		When set to **true** or **TRUE**, any routes with a wildcard policy of **Subdomain** that pass the router admission checks will be serviced by the HAProxy router.
`ROUTER_DISABLE_NAMESPACE_OWNERSHIP_CHECK`		Set to **true** to relax the namespace ownership policy.

TimeUnits are represented by a number followed by the unit **us(microseconds)**, `ms` **(milliseconds, default)**, `s` **(seconds)**, `m` **(minutes)**, `h` ***(hours)**, `d` **(days). The regular expression is: [1-9][0-9]**(us\|ms\|s\|m\|h\|d)

> **Note**
>
> If you want to run multiple routers on the same machine, you must change the ports that the router is listening on, **ROUTER_SERVICE_SNI_PORT** and **ROUTER_SERVICE_NO_SNI_PORT**. These ports can be anything you want as long as they are unique on the machine. These ports will not be exposed externally.

3.7.5. Route Host Names

In order for services to be exposed externally, an OpenShift Container Platform route allows you to associate a service with an externally-reachable host name. This edge host name is then used to route traffic to the service.

When multiple routes from different namespaces claim the same host, the oldest route wins and claims it for the namespace. If additional routes with different path fields are defined in the same namespace, those paths are added. If multiple routes with the same path are used, the oldest takes priority.

A consequence of this behavior is that if you have two routes for a host name: an older one and a newer one. If someone else has a route for the same host name that they created between when you created the other two routes, then if you delete your older route, your claim to the host name will no longer be in effect. The other namespace now claims the host name and your claim is lost.

> **Example 3.10. A Route with a Specified Host:**

```
apiVersion: v1
kind: Route
metadata:
  name: host-route
spec:
  host: www.example.com    1
  to:
    kind: Service
    name: service-name
```

1 1 1

Specifies the externally-reachable host name used to expose a service.

Example 3.11. A Route Without a Host:

```
apiVersion: v1
kind: Route
metadata:
  name: no-route-hostname
spec:
  to:
    kind: Service
    name: service-name
```

If a host name is not provided as part of the route definition, then OpenShift Container Platform automatically generates one for you. The generated host name is of the form:

```
<route-name>[-<namespace>].<suffix>
```

The following example shows the OpenShift Container Platform-generated host name for the above configuration of a route without a host added to a namespace **mynamespace**:

Example 3.12. Generated Host Name

```
no-route-hostname-mynamespace.router.default.svc.cluster.local  1
```

1

The generated host name suffix is the default routing subdomain **router.default.svc.cluster.local**.

A cluster administrator can also customize the suffix used as the default routing subdomain for their environment.

3.7.6. Route Types

Routes can be either secured or unsecured. Secure routes provide the ability to use several types of TLS termination to serve certificates to the client. Routers support edge, passthrough, and re-encryption termination.

Example 3.13. Unsecured Route Object YAML Definition

```
apiVersion: v1
kind: Route
metadata:
  name: route-unsecured
spec:
  host: www.example.com
  to:
    kind: Service
    name: service-name
```

Unsecured routes are simplest to configure, as they require no key or certificates, but secured routes offer security for connections to remain private.

A secured route is one that specifies the TLS termination of the route. The available types of termination are described below.

3.7.7. Path Based Routes

Path based routes specify a path component that can be compared against a URL (which requires that the traffic for the route be HTTP based) such that multiple routes can be served using the same host name, each with a different path. Routers should match routes based on the most specific path to the least; however, this depends on the router implementation. The following table shows example routes and their accessibility:

Table 3.2. Route Availability

Route	When Compared to	Accessible
www.example.com/test	*www.example.com/test*	Yes
	www.example.com	No

Route	When Compared to	Accessible
www.example.com/test and *www.example.com*	*www.example.com/test*	Yes
	www.example.com	Yes
www.example.com	*www.example.com/test*	Yes (Matched by the host, not the route)
	www.example.com	Yes

Example 3.14. An Unsecured Route with a Path:

```
apiVersion: v1
kind: Route
metadata:
  name: route-unsecured
spec:
  host: www.example.com
  path: "/test"    1
  to:
    kind: Service
    name: service-name
```

1

The path is the only added attribute for a path-based route.

> **Note**
>
> Path-based routing is not available when using passthrough TLS, as the router does not terminate TLS in that case and cannot read the contents of the request.

3.7.8. Secured Routes

Secured routes specify the TLS termination of the route and, optionally, provide a key and certificate(s).

> **Note**
>
> TLS termination in OpenShift Container Platform relies on SNI for serving custom certificates. Any non-SNI traffic received on port 443 is handled with TLS termination and a default certificate (which may not match the requested host name, resulting in validation errors).

Secured routes can use any of the following three types of secure TLS termination.

Edge Termination

With edge termination, TLS termination occurs at the router, prior to proxying traffic to its destination. TLS certificates are served by the front end of the router, so they must be configured into the route, otherwise the router's default certificate will be used for TLS termination.

Example 3.15. A Secured Route Using Edge Termination

```
apiVersion: v1
kind: Route
metadata:
  name: route-edge-secured 1
spec:
  host: www.example.com
  to:
    kind: Service
    name: service-name 2
  tls:
    termination: edge          3
    key: |-                    4
      -----BEGIN PRIVATE KEY-----
      [...]
      -----END PRIVATE KEY-----
    certificate: |-            5
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
    caCertificate: |-          6
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
```

1 **2**

The name of the object, which is limited to 63 characters.

3

The **termination** field is **edge** for edge termination.

4

The **key** field is the contents of the PEM format key file.

5

The **certificate** field is the contents of the PEM format certificate file.

6

An optional CA certificate may be required to establish a certificate chain for validation.

Because TLS is terminated at the router, connections from the router to the endpoints over the internal network are not encrypted.

Edge-terminated routes can specify an **insecureEdgeTerminationPolicy** that enables traffic on insecure schemes (**HTTP**) to be disabled, allowed or redirected. The allowed values for **insecureEdgeTerminationPolicy** are: **None** or empty (for disabled),**Allow** or **Redirect**. The default **insecureEdgeTerminationPolicy** is to disable traffic on the insecure scheme. A common use case is to allow content to be served via a secure scheme but serve the assets (example images, stylesheets and javascript) via the insecure scheme.

Example 3.16. A Secured Route Using Edge Termination Allowing HTTP Traffic

```
apiVersion: v1
kind: Route
metadata:
  name: route-edge-secured-allow-insecure  1
spec:
  host: www.example.com
  to:
    kind: Service
    name: service-name  2
  tls:
    termination:                      edge   3
    insecureEdgeTerminationPolicy: Allow   4
    [ ... ]
```

1 2

The name of the object, which is limited to 63 characters.

3

The **termination** field is **edge** for edge termination.

4

The insecure policy to allow requests sent on an insecure scheme **HTTP**.

Example 3.17. A Secured Route Using Edge Termination Redirecting HTTP Traffic to HTTPS

```
apiVersion: v1
kind: Route
metadata:
  name: route-edge-secured-redirect-insecure  1
spec:
  host: www.example.com
  to:
    kind: Service
    name: service-name  2
  tls:
    termination:                      edge        3
    insecureEdgeTerminationPolicy: Redirect  4
    [ ... ]
```

1 2

The name of the object, which is limited to 63 characters.

3

The **termination** field is **edge** for edge termination.

4

The insecure policy to redirect requests sent on an i/nsecure scheme **HTTP** to a secure scheme **HTTPS**.

Passthrough Termination

With passthrough termination, encrypted traffic is sent straight to the destination without the router providing TLS termination. Therefore no key or certificate is required.

Example 3.18. A Secured Route Using Passthrough Termination

```
apiVersion: v1
kind: Route
metadata:
  name: route-passthrough-secured 1
spec:
  host: www.example.com
  to:
    kind: Service
    name: service-name 2
  tls:
    termination: passthrough 3
```

1 **2**

The name of the object, which is limited to 63 characters.

3

The **termination** field is set to **passthrough**. No other encryption fields are needed.

The destination pod is responsible for serving certificates for the traffic at the endpoint. This is currently the only method that can support requiring client certificates (also known as two-way authentication).

> **Note**
>
> passthrough routes can also have an **insecureEdgeTerminationPolicy** the only valid values are **None** or empty (for disabled) or **Redirect**.

Re-encryption Termination

Re-encryption is a variation on edge termination where the router terminates TLS with a certificate, then re-encrypts its connection to the endpoint which may have a different certificate. Therefore the full path of the connection is encrypted, even over the internal network. The router uses health checks to determine the authenticity of the host.

Example 3.19. A Secured Route Using Re-Encrypt Termination

```
apiVersion: v1
kind: Route
metadata:
  name: route-pt-secured 1
spec:
  host: www.example.com
  to:
    kind: Service
```

```
    name: service-name      2
  tls:
    termination: reencrypt          3
    key: [as in edge termination]
    certificate: [as in edge termination]
    caCertificate: [as in edge termination]
    destinationCACertificate: |-     4
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
```

1 2

The name of the object, which is limited to 63 characters.

3

The **termination** field is set to **reencrypt**. Other fields are as in edge termination.

4

The **destinationCACertificate** field specifies a CA certificate to validate the endpoint certificate, securing the connection from the router to the destination. This field is required, but only for re-encryption.

3.7.9. Router Sharding

In OpenShift Container Platform, each route can have any number of labels in its **metadata** field. A router uses *selectors* (also known as a *selection expression*) to select a subset of routes from the entire pool of routes to serve. A selection expression can also involve labels on the route's namespace. The selected routes form a *router shard*. You can create and modify router shards independently from the routes, themselves.

This design supports *traditional* sharding as well as *overlapped* sharding. In traditional sharding, the selection results in no overlapping sets and a route belongs to exactly one shard. In overlapped sharding, the selection results in overlapping sets and a route can belong to many different shards. For example, a single route may belong to a **SLA=high** shard (but not **SLA=medium** or **SLA=low** shards), as well as a **geo=west** shard (but not a **geo=east** shard).

Another example of overlapped sharding is a set of routers that select based on namespace of the route:

Router	Selection	Namespaces
router-1	**A* — J***	**A***, **B***, **C***, **D***, **E***, **F***, **G***, **H***, **I***, **J***

Router	Selection	Namespaces
router-2	**K* ─ T***	**K***, **L***, **M***, **N***, **O***, **P***, **Q***, **R***, **S***, **T***
router-3	**Q* ─ Z***	**Q***, **R***, **S***, **T***, **U***, **V***, **W***, **X***, **Y***, **Z***

Both **router-2** and **router-3** serve routes that are in the namespaces **Q***, **R***, **S***, **T***. To change this example from overlapped to traditional sharding, we could change the selection of **router-2** to **K* ─ P***, which would eliminate the overlap.

When routers are sharded, a given route is bound to zero or more routers in the group. The route binding ensures uniqueness of the route across the shard. Uniqueness allows secure and non-secure versions of the same route to exist within a single shard. This implies that routes now have a visible life cycle that moves from created to bound to active.

In the sharded environment the first route to hit the shard reserves the right to exist there indefinitely, even across restarts.

During a green/blue deployment a route may be be selected in multiple routers. An OpenShift Container Platform application administrator may wish to bleed traffic from one version of the application to another and then turn off the old version.

Sharding can be done by the administrator at a cluster level and by the user at a project/namespace level. When namespace labels are used, the service account for the router must have **cluster-reader** permission to permit the router to access the labels in the namespace.

> **Note**
>
> For two or more routes that claim the same host name, the resolution order is based on the age of the route and the oldest route would win the claim to that host. In the case of sharded routers, routes are selected based on their labels matching the router's selection criteria. There is no consistent way to determine when labels are added to a route. So if an older route claiming an existing host name is "re-labelled" to match the router's selection criteria, it will replace the existing route based on the above mentioned resolution order (oldest route wins).

3.7.10. Route-specific Annotations

Using environment variables as defined in Configuration Parameters, 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 3.3. Route Annotations

Variable	Description	Environment Variable Used as Default
`haproxy.router.openshift.io/balance`	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**.
`haproxy.router.openshift.io/disable_cookies`	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.	
`haproxy.router.openshift.io/rate-limit-connections`	Setting **true** or **TRUE** to enables rate limiting functionality.	
`haproxy.router.openshift.io/rate-limit-connections.concurrent-tcp`	Limits the number of concurrent TCP connections shared by an IP address.	
`haproxy.router.openshift.io/rate-limit-connections.rate-http`	Limits the rate at which an IP address can make HTTP requests.	
`haproxy.router.openshift.io/rate-limit-connections.rate-tcp`	Limits the rate at which an IP address can make TCP connections.	
`haproxy.router.openshift.io/timeout`	Sets a server-side timeout for the route. (TimeUnits)	**ROUTER_DEFAULT_SERVER_TIMEOUT**

Variable	Description	Environment Variable Used as Default
router.openshift.io/haproxy.health.check.interval	Sets the interval for the back-end health checks. (TimeUnits)	**ROUTER_BACKEND_CHECK_INTERVAL**

Example 3.20. A Route Setting Custom Timeout

```
apiVersion: v1
kind: Route
metadata:
  annotations:
    haproxy.router.openshift.io/timeout: 5500ms  1
[...]
```

1

> 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.

3.7.11. Creating Routes Specifying a Wildcard Subdomain Policy

A wildcard policy allows a user to define a route that covers all hosts within a domain (when the router is configured to allow it). A route can specify a wildcard policy as part of its configuration using the **wildcardPolicy** field. Any routers run with a policy allowing wildcard routes will expose the route appropriately based on the wildcard policy.

Learn how to configure HAProxy routers to allow wildcard routes.

A Route Specifying a Subdomain WildcardPolicy

```
apiVersion: v1
kind: Route
spec:
  host: wildcard.example.com  1
```

```
wildcardPolicy: Subdomain      2
to:
  kind: Service
  name: service-name
```

1

Specifies the externally reachable host name used to expose a service.

2

Specifies that the externally reachable host name should allow all hosts in the subdomain **example.com**. ***.example.com** is the subdomain for host name **wildcard.example.com** to reach the exposed service.

3.7.12. Route Status

The **route status** field is only set by routers. If changes are made to a route so that a router no longer serves a specific route, the status becomes stale. The routers do not clear the **route status** field. To remove the stale entries in the route status, use the clear-route-status script.

3.7.13. Denying or Allowing Certain Domains in Routes

A router can be configured to deny or allow a specific subset of domains from the host names in a route using the **ROUTER_DENIED_DOMAINS** and **ROUTER_ALLOWED_DOMAINS** environment variables.

ROUTER_DENIED_DOMAINS	Domains listed are not allowed in any indicated routes.
ROUTER_ALLOWED_DOMAINS	Only the domains listed are allowed in any indicated routes.

The domains in the list of denied domains take precedence over the list of allowed domains. Meaning OpenShift Container Platform first checks the deny list (if applicable), and if the host name is not in the list of denied domains, it then checks the list of allowed domains. However, the list of allowed domains is more restrictive, and ensures that the router only admits routes with hosts that belong to that list.

For example, to deny the **[*.]open.header.test**, **[*.]openshift.org** and **[*.]block.it** routes for the **myrouter** route:

```
$ oadm router myrouter ...
$ oc set env dc/myrouter ROUTER_DENIED_DOMAINS="open.header.test,
openshift.org, block.it"
```

This means that **myrouter** will admit the following based on the route's name:

```
$ oc expose service/<name> --hostname="foo.header.test"
$ oc expose service/<name> --hostname="www.allow.it"
$ oc expose service/<name> --hostname="www.openshift.test"
```

However, **myrouter** will deny the following:

```
$ oc expose service/<name> --hostname="open.header.test"
$ oc expose service/<name> --hostname="www.open.header.test"
$ oc expose service/<name> --hostname="block.it"
$ oc expose service/<name> --hostname="franco.baresi.block.it"
$ oc expose service/<name> --hostname="openshift.org"
$ oc expose service/<name> --hostname="api.openshift.org"
```

Alternatively, to block any routes where the host name is *not* set to
[*.]stickshift.org or **[*.]kates.net**:

```
$ oadm router myrouter ...
$ oc set env dc/myrouter ROUTER_ALLOWED_DOMAINS="stickshift.org,
kates.net"
```

This means that the **myrouter** router will admit:

```
$ oc expose service/<name> --hostname="stickshift.org"
$ oc expose service/<name> --hostname="www.stickshift.org"
$ oc expose service/<name> --hostname="kates.net"
$ oc expose service/<name> --hostname="api.kates.net"
$ oc expose service/<name> --hostname="erno.r.kube.kates.net"
```

However, **myrouter** will deny the following:

```
$ oc expose service/<name> --hostname="www.open.header.test"
$ oc expose service/<name> --hostname="drive.ottomatic.org"
$ oc expose service/<name> --hostname="www.wayless.com"
$ oc expose service/<name> --hostname="www.deny.it"
```

To implement both scenarios, run:

```
$ oadm router adrouter ...
$ oc env dc/adrouter ROUTER_ALLOWED_DOMAINS="openshift.org, kates.net"
\
    ROUTER_DENIED_DOMAINS="ops.openshift.org, metrics.kates.net"
```

This will allow any routes where the host name is set to **[*.]openshift.org** or
[*.]kates.net, and not allow any routes where the host name is set to
[*.]ops.openshift.org or **[*.]metrics.kates.net**.

Therefore, the following will be denied:

```
$ oc expose service/<name> --hostname="www.open.header.test"
$ oc expose service/<name> --hostname="ops.openshift.org"
$ oc expose service/<name> --hostname="log.ops.openshift.org"
$ oc expose service/<name> --hostname="www.block.it"
```

```
$ oc expose service/<name> --hostname="metrics.kates.net"
$ oc expose service/<name> --hostname="int.metrics.kates.net"
```

However, the following will be allowed:

```
$ oc expose service/<name> --hostname="openshift.org"
$ oc expose service/<name> --hostname="api.openshift.org"
$ oc expose service/<name> --hostname="m.api.openshift.org"
$ oc expose service/<name> --hostname="kates.net"
$ oc expose service/<name> --hostname="api.kates.net"
```

3.8. TEMPLATES

3.8.1. Overview

A template describes a set of objects that can be parameterized and processed to produce a list of objects for creation by OpenShift Container Platform. 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. NETWORKING

4.1.1. 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. However, it is not recommended that you have a pod talk to another directly by using the IP address. Instead, we recommend that you create a service, then interact with the service.

4.1.2. OpenShift Container Platform 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 IP, 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 Container Platform has a built-in DNS so that the services can be reached by the service DNS as well as the service IP/port. OpenShift Container Platform 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 4.1. DNS Example Names

Object Type	Example
Default	<pod_namespace>.cluster.local
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 creates a new IP for the service. This also removes the need to use environment variables, as pods can use the service DNS. Also, as the DNS does not change, you can reference database services as **db.local** in config files. Wildcard lookups are also supported, as 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.

4.1.3. Network Plug-ins

OpenShift Container Platform supports the same plug-in model as Kubernetes for networking pods. The following network plug-ins are currently supported by OpenShift Container Platform.

4.1.3.1. OpenShift SDN

OpenShift Container Platform deploys a software-defined networking (SDN) approach for connecting pods in an OpenShift Container Platform cluster. The OpenShift SDN connects all pods across all node hosts, providing a unified cluster network.

OpenShift SDN is automatically installed and configured as part of the Ansible-based installation procedure. See the OpenShift Container Platform SDN section for more information.

4.1.3.2. Nuage SDN for OpenShift Container Platform

Nuage Networks' SDN solution delivers highly scalable, policy-based overlay networking for pods in an OpenShift Container Platform cluster. Nuage SDN can be installed and configured as a part of the Ansible-based installation procedure. See the Advanced Installation section for information on how to install and deploy OpenShift Container Platform with Nuage SDN.

Nuage Networks provides a highly scalable, policy-based SDN platform called Virtualized Services Platform (VSP). Nuage VSP uses an SDN Controller, along with the open source Open vSwitch for the data plane.

Nuage uses overlays to provide policy-based networking between OpenShift Container

Platform and other environments consisting of VMs and bare metal servers. The platform's real-time analytics engine enables visibility and security monitoring for OpenShift Container Platform applications.

Nuage VSP integrates with OpenShift Container Platform to allows business applications to be quickly turned up and updated by removing the network lag faced by DevOps teams.

Figure 4.1. Nuage VSP Integration with OpenShift Container Platform
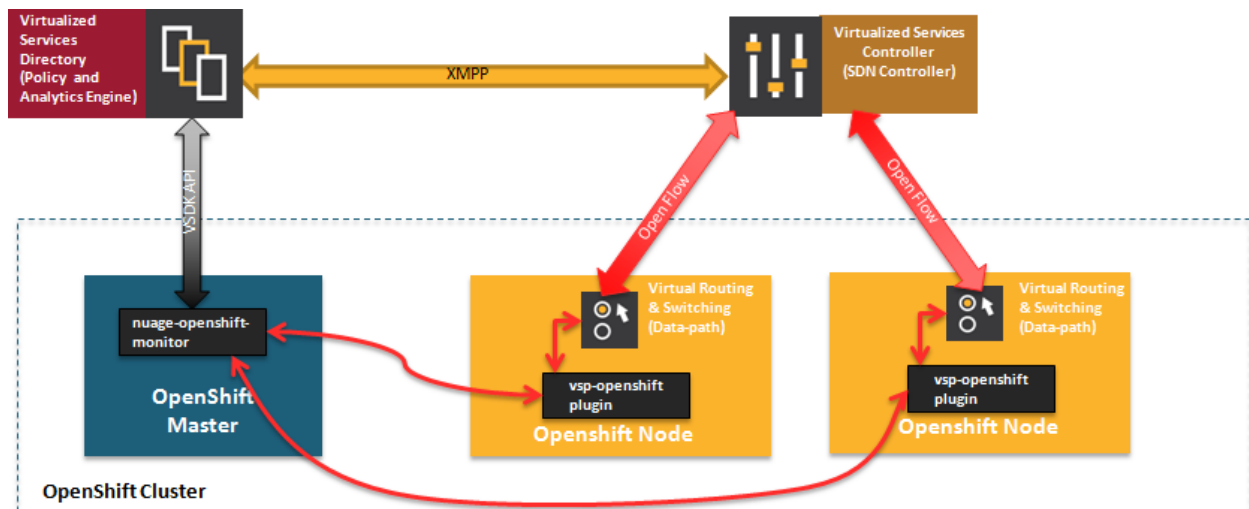
There are two specific components responsible for the integration.

1. The **nuage-openshift-monitor** service, which runs as a separate service on the OpenShift Container Platform master node.

2. The **vsp-openshift** plug-in, which is invoked by the OpenShift Container Platform runtime on each of the nodes of the cluster.

Nuage Virtual Routing and Switching software (VRS) is based on open source Open vSwitch and is responsible for the datapath forwarding. The VRS runs on each node and gets policy configuration from the controller.

Nuage VSP Terminology

Figure 4.2. Nuage VSP Building Blocks
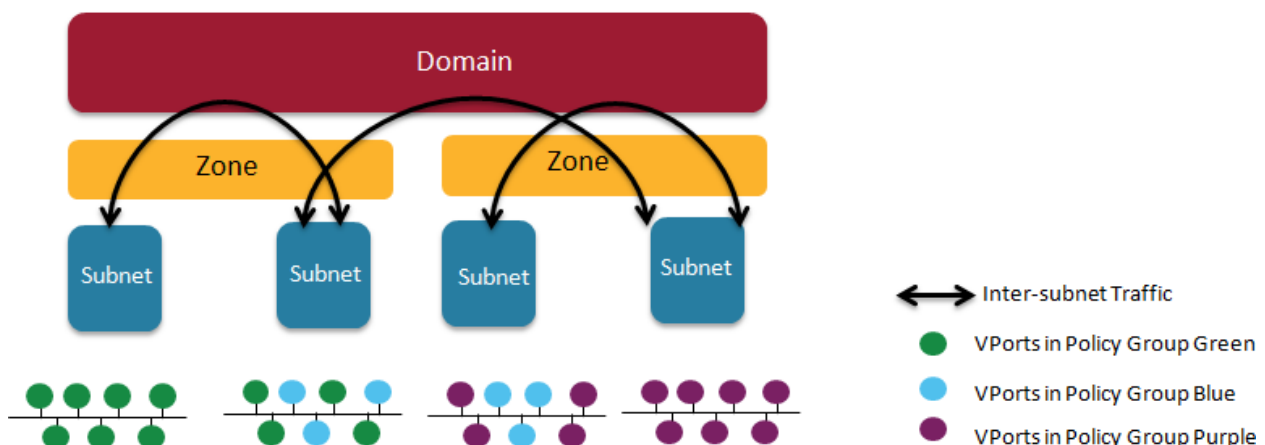
1. Domains: An organization contains one or more domains. A domain is a single "Layer 3" space. In standard networking terminology, a domain maps to a VRF instance.

2. Zones: Zones are defined under a domain. A zone does not map to anything on the network directly, but instead acts as an object with which policies are associated such that all endpoints in the zone adhere to the same set of policies.

3. Subnets: Subnets are defined under a zone. A subnet is a specific Layer 2 subnet within the domain instance. A subnet is unique and distinct within a domain, that is, subnets within a Domain are not allowed to overlap or to contain other subnets in accordance with the standard IP subnet definitions.

4. VPorts: A VPort is a new level in the domain hierarchy, intended to provide more granular configuration. In addition to containers and VMs, VPorts are also used to attach Host and Bridge Interfaces, which provide connectivity to Bare Metal servers, Appliances, and Legacy VLANs.

5. Policy Group: Policy Groups are collections of VPorts.

Mapping of Constructs

Many OpenShift Container Platform concepts have a direct mapping to Nuage VSP constructs:

Figure 4.3. Nuage VSP and OpenShift Container Platform mapping
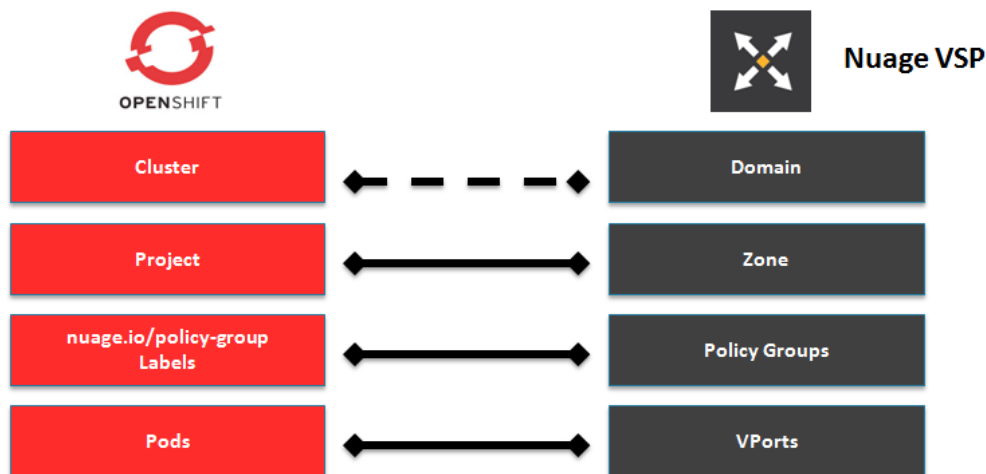
A Nuage subnet is not mapped to an OpenShift Container Platform node, but a subnet for a particular project can span multiple nodes in OpenShift Container Platform.

A pod spawning in OpenShift Container Platform translates to a virtual port being created in VSP. The **vsp-openshift** plug-in interacts with the VRS and gets a policy for that virtual port from the VSD via the VSC. Policy Groups are supported to group multiple pods together that must have the same set of policies applied to them.

Currently, pods can only be assigned to policy groups using the operations workflow where a policy group is created by the administrative user in VSD. The pod being a part of the policy group is specified by means of **nuage.io/policy-group** label in the specification of the pod.

4.1.3.2.1. Integration Components

Nuage VSP integrates with OpenShift Container Platform using two main components:

1. **nuage-openshift-monitor**

2. **vsp-openshift plugin**

nuage-openshift-monitor

nuage-openshift-monitor is a service that monitors the OpenShift Container Platform API server for creation of projects, services, users, user-groups, etc.

> **Note**
>
> In case of a Highly Available (HA) OpenShift Container Platform cluster with multiple masters, **nuage-openshift-monitor** process runs on all the masters independently without any change in functionality.

For the developer workflow, **nuage-openshift-monitor** also auto-creates VSD objects by exercising the VSD REST API to map OpenShift Container Platform constructs to VSP constructs. Each cluster instance maps to a single domain in Nuage VSP. This allows a given enterprise to potentially have multiple cluster installations - one per domain instance for that Enterprise in Nuage. Each OpenShift Container Platform project is mapped to a zone in the domain of the cluster on the Nuage VSP. Whenever **nuage-openshift-monitor** sees an addition or deletion of the project, it instantiates a zone using the VSDK APIs corresponding to that project and allocates a block of subnet for that zone. Additionally, the **nuage-openshift-monitor** also creates a network macro group for this project. Likewise, whenever **nuage-openshift-monitor** sees an addition ordeletion of a service, it creates a network macro corresponding to the service IP and assigns that network macro to the network macro group for that project (user provided network macro group using labels is also supported) to enable communication to that service.

For the developer workflow, all pods that are created within the zone get IPs from that subnet pool. The subnet pool allocation and management is done by **nuage-openshift-monitor** based on a couple of plug-in specific parameters in the master-config file. However the actual IP address resolution and vport policy resolution is still done by VSD based on the domain/zone that gets instantiated when the project is created. If the initial subnet pool is exhausted, **nuage-openshift-monitor** carves out an additional subnet from the cluster CIDR to assign to a given project.

For the operations workflow, the users specify Nuage recognized labels on their application or pod specification to resolve the pods into specific user-defined zones and subnets. However, this cannot be used to resolve pods in the zones or subnets created via the developer workflow by **nuage-openshift-monitor**.

Note

In the operations workflow, the administrator is responsible for pre-creating the VSD constructs to map the pods into a specific zone/subnet as well as allow communication between OpenShift entities (ACL rules, policy groups, network macros, and network macro groups). Detailed description of how to use Nuage labels is provided in the Nuage VSP Openshift Integration Guide.

vsp-openshift Plug-in

The vsp-openshift networking plug-in is called by the OpenShift Container Platform runtime on each OpenShift Container Platform node. It implements the network plug-in init and pod setup, teardown, and status hooks. The vsp-openshift plug-in is also responsible for allocating the IP address for the pods. In particular, it communicates with the VRS (the forwarding engine) and configures the IP information onto the pod.

4.2. OPENSHIFT SDN

4.2.1. Overview

OpenShift Container Platform uses a software-defined networking (SDN) approach to provide a unified cluster network that enables communication between pods across the OpenShift Container Platform cluster. This pod network is established and maintained by the OpenShift SDN, which configures an overlay network using Open vSwitch (OVS).

OpenShift SDN provides two SDN plug-ins for configuring the pod network:

» The **ovs-subnet** plug-in is the original plug-in which provides a "flat" pod network where every pod can communicate with every other pod and service.

» The **ovs-multitenant** plug-in provides OpenShift Container Platform project level isolation for pods and services. Each project receives a unique Virtual Network ID (VNID) that identifies traffic from pods assigned to the project. Pods from different projects cannot send packets to or receive packets from pods and services of a different project.

However, projects which receive VNID 0 are more privileged in that they are allowed to communicate with all other pods, and all other pods can communicate with them. In OpenShift Container Platform clusters, the **default** project has VNID 0. This facilitates certain services like the load balancer, etc. to communicate with all other pods in the cluster and vice versa.

Following is a detailed discussion of the design and operation of OpenShift SDN, which may be useful for troubleshooting.

Note

Information on configuring the SDN on masters and nodes is available in Configuring the SDN.

4.2.2. Design on Masters

On an OpenShift Container Platform master, OpenShift SDN maintains a registry of nodes, stored in **etcd**. When the system administrator registers a node, OpenShift SDN allocates an unused subnet from the cluster network and stores this subnet in the registry. When a node is deleted, OpenShift SDN deletes the subnet from the registry and considers the subnet available to be allocated again.

In the default configuration, the cluster network is the **10.128.0.0/14** network (i.e. **10.128.0.0** - **10.131.255.255**), and nodes are allocated **/23** subnets (i.e., **10.128.0.0/23**, **10.128.2.0/23**, **10.128.4.0/23**, and so on). This means that the cluster network has 512 subnets available to assign to nodes, and a given node is allocated 510 addresses that it can assign to the containers running on it. The size and address range of the cluster network are configurable, as is the host subnet size.

Note that OpenShift SDN on a master does not configure the local (master) host to have access to any cluster network. Consequently, a master host does not have access to pods via the cluster network, unless it is also running as a node.

When using the **ovs-multitenant** plug-in, the OpenShift SDN master also watches for the creation and deletion of projects, and assigns VXLAN VNIDs to them, which will be used later by the nodes to isolate traffic correctly.

4.2.3. Design on Nodes

On a node, OpenShift SDN first registers the local host with the SDN master in the aforementioned registry so that the master allocates a subnet to the node.

Next, OpenShift SDN creates and configures three network devices:

- » **br0**, the OVS bridge device that pod containers will be attached to. OpenShift SDN also configures a set of non-subnet-specific flow rules on this bridge.

- » **tun0**, an OVS internal port (port 2 on**br0**). This gets assigned the cluster subnet gateway address, and is used for external network access. OpenShift SDN configures **netfilter** and routing rules to enable access from the cluster subnet to the external network via NAT.

- » **vxlan0**, the OVS VXLAN device (port 1 on**br0**), which provides access to containers on remote nodes.

Each time a pod is started on the host, OpenShift SDN:

1. assigns the pod a free IP address from the node's cluster subnet.

2. attaches the host side of the pod's veth interface pair to the OVS bridge **br0**.

3. adds OpenFlow rules to the OVS database to route traffic addressed to the new pod to the correct OVS port.

4. in the case of the **ovs-multitenant** plug-in, adds OpenFlow rules to tag traffic coming from the pod with the pod's VNID, and to allow traffic into the pod if the traffic's VNID matches the pod's VNID (or is the privileged VNID 0). Non-matching traffic is filtered out by a generic rule.

OpenShift SDN nodes also watch for subnet updates from the SDN master. When a new subnet is added, the node adds OpenFlow rules on **br0** so that packets with a destination IP address the remote subnet go to **vxlan0** (port 1 on **br0**) and thus out onto the network. The **ovs-subnet** plug-in sends all packets across the VXLAN with

VNID 0, but the **ovs-multitenant** plug-in uses the appropriate VNID for the source container.

4.2.4. Packet Flow

Suppose you have two containers, A and B, where the peer virtual Ethernet device for container A's **eth0** is named **vethA** and the peer for container B's **eth0** is named **vethB**.

> **Note**
>
> If the Docker service's use of peer virtual Ethernet devices is not already familiar to you, review Docker's advanced networking documentation.

Now suppose first that container A is on the local host and container B is also on the local host. Then the flow of packets from container A to container B is as follows:

eth0 (in A's netns) →*vethA* → *br0* → *vethB* →*eth0* (in B's netns)

Next, suppose instead that container A is on the local host and container B is on a remote host on the cluster network. Then the flow of packets from container A to container B is as follows:

eth0 (in A's netns) →*vethA* → *br0* → *vxlan0* → network [1] → *vxlan0* → *br0* →*vethB* → *eth0* (in B's netns)

Finally, if container A connects to an external host, the traffic looks like:

eth0 (in A's netns) →*vethA* → *br0* → *tun0* → (NAT) →*eth0* (physical device) → Internet

Almost all packet delivery decisions are performed with OpenFlow rules in the OVS bridge **br0**, which simplifies the plug-in network architecture and provides flexible routing. In the case of the **ovs-multitenant** plug-in, this also provides enforceable network isolation.

4.2.5. Network Isolation

You can use the **ovs-multitenant** plug-in to achieve network isolation. When a packet exits a pod assigned to a non-default project, the OVS bridge **br0** tags that packet with the project's assigned VNID. If the packet is directed to another IP address in the node's cluster subnet, the OVS bridge only allows the packet to be delivered to the destination pod if the VNIDs match.

If a packet is received from another node via the VXLAN tunnel, the Tunnel ID is used as the VNID, and the OVS bridge only allows the packet to be delivered to a local pod if the tunnel ID matches the destination pod's VNID.

Packets destined for other cluster subnets are tagged with their VNID and delivered to the VXLAN tunnel with a tunnel destination address of the node owning the cluster subnet.

As described before, VNID 0 is privileged in that traffic with any VNID is allowed to enter any pod assigned VNID 0, and traffic with VNID 0 is allowed to enter any pod. Only the **default** OpenShift Container Platform project is assigned VNID 0; all other projects are assigned unique, isolation-enabled VNIDs. Cluster administrators can

optionally control the pod network for the project using the administrator CLI.

4.3. FLANNEL

4.3.1. Overview

flannel is a virtual networking layer designed specifically for containers. OpenShift Container Platform can use it for networking containers instead of the default software-defined networking (SDN) components. This is useful if running OpenShift Container Platform within a cloud provider platform that also relies on SDN, such as OpenStack, and you want to avoid encapsulating packets twice through both platforms.

4.3.2. Architecture

OpenShift Container Platform runs **flannel** in **host-gw** mode, which maps routes from container to container. Each host within the network runs an agent called **flanneld**, which is responsibile for:

- Managing a unique subnet on each host

- Distributing IP addresses to each container on its host

- Mapping routes from one container to another, even if on different hosts

Each **flanneld** agent provides this infomation to a centralized**etcd** store so other agents on hosts can route packets to other containers within the **flannel** network.

The following diagram illustrates the architecture and data flow from one container to another using a **flannel** network:
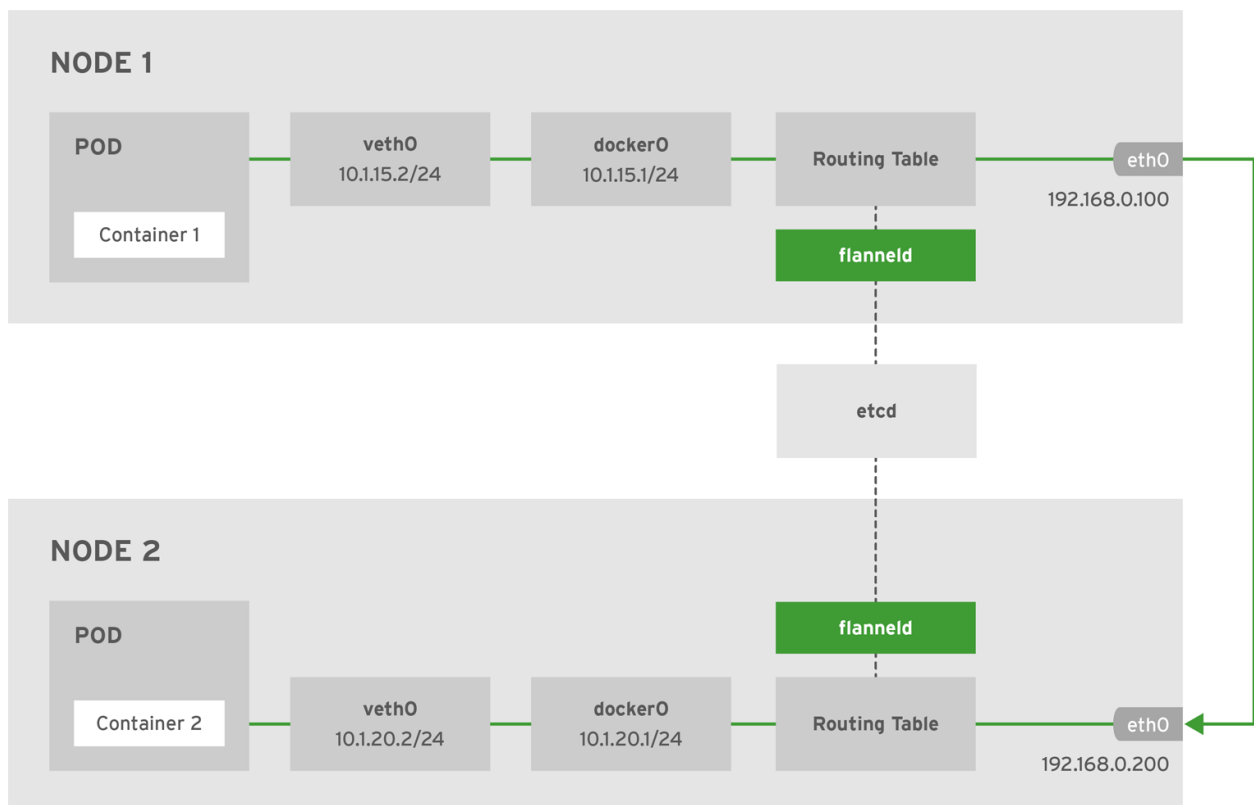
OPENSHIFT_436034_0317

Node 1 would contain the following routes:

```
default via 192.168.0.100 dev eth0 proto static metric 100
10.1.15.0/24 dev docker0 proto kernel scope link src 10.1.15.1
10.1.20.0/24 via 192.168.0.200 dev eth0
```

Node 2 would contain the following routes:

```
default via 192.168.0.200 dev eth0 proto static metric 100
10.1.20.0/24 dev docker0 proto kernel scope link src 10.1.20.1
10.1.15.0/24 via 192.168.0.100 dev eth0
```

4.4. F5 BIG-IP® ROUTER PLUG-IN

4.4.1. Overview

> **Note**
>
> The F5 router plug-in is available starting in OpenShift Enterprise 3.0.2.

The F5 router plug-in integrates with an existing **F5 BIG-IP®** system in your environment. **F5 BIG-IP®** version 11.4 or newer is required in order to have the F5 iControl REST API. The F5 router supports unsecured, edge terminated, re-encryption terminated, and passthrough terminated routes matching on HTTP vhost and request path.

The F5 router has additional features over the **F5 BIG-IP®** support in OpenShift Enterprise 2. Compared with the **routing-daemon** used in earlier versions, the F5 router additionally supports:

» path-based routing (using policy rules),

» passthrough of encrypted connections (implemented using an iRule that parses the SNI protocol and uses a data group that is maintained by the F5 router for the servername lookup).

> **Note**
>
> Passthrough routes are a special case: path-based routing is technically impossible with passthrough routes because **F5 BIG-IP®** itself does not see the HTTP request, so it cannot examine the path. The same restriction applies to the template router; it is a technical limitation of passthrough encryption, not a technical limitation of OpenShift Container Platform.

4.4.2. Routing Traffic to Pods Through the SDN

Because **F5 BIG-IP®** is external to the OpenShift SDN, a cluster administrator must create a peer-to-peer tunnel between **F5 BIG-IP®** and a host that is on the SDN, typically an OpenShift Container Platform node host. This *ramp node* can be configured

as unschedulable for pods so that it will not be doing anything except act as a gateway for the **F5 BIG-IP®** host. It is also possible to configure multiple such hosts and use the OpenShift Container Platform **ipfailover** feature for redundancy; the **F5 BIG-IP®** host would then need to be configured to use the **ipfailover** VIP for its tunnel's remote endpoint.

4.4.3. F5 Integration Details

The operation of the F5 router is similar to that of the OpenShift Container Platform **routing-daemon** used in earlier versions. Both use REST API calls to:

≫ create and delete pools,

≫ add endpoints to and delete them from those pools, and

≫ configure policy rules to route to pools based on vhost.

Both also use **scp** and **ssh** commands to upload custom TLS/SSL certificates to **F5 BIG-IP®**.

The F5 router configures pools and policy rules on virtual servers as follows:

≫ When a user creates or deletes a route on OpenShift Container Platform, the router creates a pool to **F5 BIG-IP®** for the route (if no pool already exists) and adds a rule to, or deletes a rule from, the policy of the appropriate vserver: the HTTP vserver for non-TLS routes, or the HTTPS vserver for edge or re-encrypt routes. In the case of edge and re-encrypt routes, the router also uploads and configures the TLS certificate and key. The router supports host- and path-based routes.

> **Note**
>
> Passthrough routes are a special case: to support those, it is necessary to write an iRule that parses the SNI ClientHello handshake record and looks up the servername in an F5 data-group. The router creates this iRule, associates the iRule with the vserver, and updates the F5 data-group as passthrough routes are created and deleted. Other than this implementation detail, passthrough routes work the same way as other routes.

≫ When a user creates a service on OpenShift Container Platform, the router adds a pool to **F5 BIG-IP®** (if no pool already exists). As endpoints on that service are created and deleted, the router adds and removes corresponding pool members.

≫ When a user deletes the route and all endpoints associated with a particular pool, the router deletes that pool.

4.4.3.1. F5 Native Integration

With native integration of F5 with OpenShift Container Platform, you do not need to configure a ramp node for F5 to be able to reach the pods on the overlay network as created by OpenShift SDN.

4.4.3.1.1. Connection

The F5 appliance can connect to the OpenShift Container Platform cluster via an L3

connection. An L2 switch connectivity is not required between OpenShift Container Platform nodes. On the appliance, you can use multiple interfaces to manage the integration:

- Management interface - Reaches the web console of the F5 appliance.

- External interface - Configures the virtual servers for inbound web traffic.

- Internal interface - Programs the appliance and reaches out to the pods.
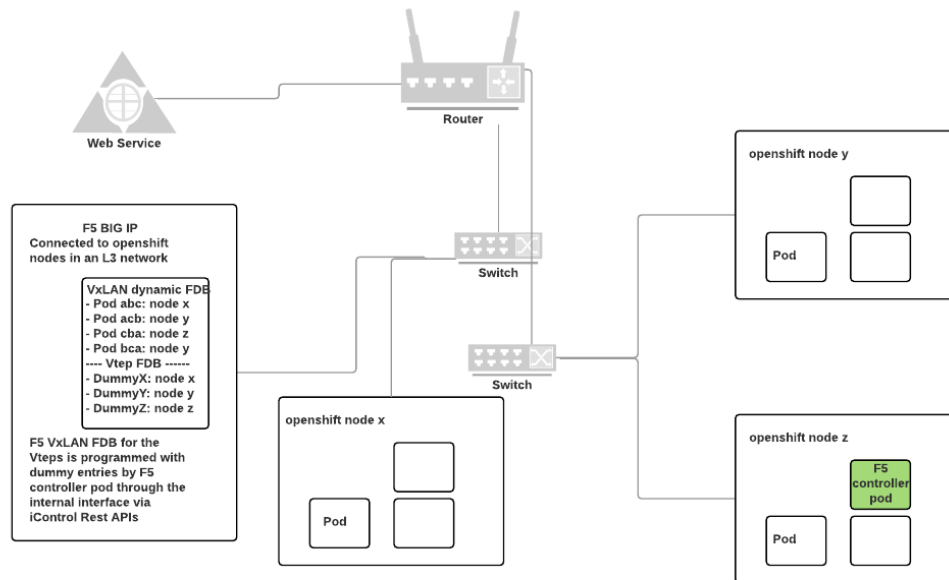
F5 + OPENSHIFT, CONNECTION DIAGRAM

An F5 controller pod has **admin** access to the appliance. The F5 image is launched within the OpenShift Container Platform cluster (scheduled on any node) that uses iControl REST APIs to program the virtual servers with policies, and configure the VxLAN device.

4.4.3.1.2. Data Flow: Packets to Pods

> **Note**
>
> This section explains how the packets reach the pods, and vice versa. These actions are performed by the F5 controller pod and the F5 appliance, not the user.

When natively integrated, The F5 appliance reaches out to the pods directly using VxLAN encapsulation. This integration works only when OpenShift Container Platform is using **openshift-sdn** as the network plug-in. The**openshift-sdn** plug-in employs VxLAN encapsulation for the overlay network that it creates.

To make a successful data path between a pod and the F5 appliance:

1. F5 needs to encapsulate the VxLAN packet meant for the pods. This requires the **sdn-services** license add-on. A VxLAN device needs to be created and the pod overlay network needs to be routed through this device.

2. F5 needs to know the VTEP IP address of the pod, which is the IP address of the node where the pod is located.

3. F5 needs to know which **source-ip** to use for the overlay network when encapsulating the packets meant for the pods. This is known as the *gateway address*.

4. OpenShift Container Platform nodes need to know where the F5 gateway address is (the VTEP address for the return traffic). This needs to be the internal interface's address. All nodes of the cluster must learn this automatically.

5. Since the overlay network is multi-tenant aware, F5 must use a VxLAN ID that is representative of an **admin** domain, ensuring that all tenants are reachable by the F5. Ensure that F5 encapsulates all packets with a **vnid** of **0** (the default **vnid** for the **admin** namespace in OpenShift Container Platform) by putting an annotation on the manually created **hostsubnet** - **pod.network.openshift.io/fixed-vnid-host: 0**.

A ghost **hostsubnet** is manually created as part of the setup, which fulfills the third and forth listed requirements. When the F5 controller pod is launched, this new ghost **hostsubnet** is provided so that the F5 appliance can be programmed suitably.
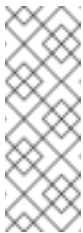
> **Note**
>
> The term *ghost* **hostsubnet** is used because it suggests that a subnet has been given to a node of the cluster. However, in reality, it is not a real node of the cluster. It is hijacked by an external appliance.

The first requirement is fulfilled by the F5 controller pod once it is launched. The second requirement is also fulfilled by the F5 controller pod, but it is an ongoing process. For each new node that is added to the cluster, the controller pod creates an entry in the VxLAN device's VTEP FDB. The controller pod needs access to the **nodes** resource in the cluster, which you can accomplish by giving the service account appropriate privileges. Use the following command:

```
$ oadm policy add-cluster-role-to-user system:sdn-reader
system:serviceaccount:default:router
```

4.4.3.1.3. Data Flow from the F5 Host

> **Note**
>
> These actions are performed by the F5 controller pod and the F5 appliance, not the user.

1. The destination pod is identified by the F5 virtual server for a packet.

2. VxLAN dynamic FDB is looked up with pod's IP address. If a MAC address is found, go to step 5.

3. Flood all entries in the VTEP FDB with ARP requests seeking the pod's MAC address.

4. One of the nodes (VTEP) will respond, confirming that it is the one where the pod is located. An entry is made into the VxLAN dynamic FDB with the pod's MAC address and the VTEP to be used as the value.

5. Encap an IP packet with VxLAN headers, where the MAC of the pod and the VTEP of the node is given as values from the VxLAN dynamic FDB.

6. Calculate the VTEP's MAC address by sending out an ARP or checking the host's neighbor cache.

7. Deliver the packet through the F5 host's internal address.

4.4.3.1.4. Data Flow: Return Traffic to the F5 Host

Note

These actions are performed by the F5 controller pod and the F5 appliance, not the user.

1. The pod sends back a packet with the destination as the F5 host's VxLAN gateway address.

2. The **openvswitch** at the node determines that the VTEP for this packet is the F5 host's internal interface address. This is learned from the ghost **hostsubnet** creation.

3. A VxLAN packet is sent out to the internal interface of the F5 host.

Note

During the entire data flow, the VNID is pre-fixed to be **0** to bypass multi-tenancy.

4.5. AUTHENTICATION

4.5.1. Overview

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

As an administrator, you can configure authentication using a master configuration file.

4.5.2. Users and Groups

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

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. These can be seen when viewing cluster bindings.

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.

4.5.3. API Authentication

Requests to the OpenShift Container Platform API are authenticated using the following methods:

OAuth Access Tokens

> ▸ Obtained from the OpenShift Container Platform OAuth server using the **_<master>_/oauth/authorize** and **_<master>_/oauth/token** endpoints.
>
> ▸ Sent as an **Authorization: Bearer…** header or an **access_token=…** query parameter

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.

4.5.3.1. Impersonation

A request to the OpenShift Container Platform API may include an **Impersonate-User** header, which indicates that the requester wants to have the request handled as though it came from the specified user. This can be done on the command line by passing the **--as=username** flag.

Before User A is allowed to impersonate User B, User A is first 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 Container Platform checks to ensure 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 are allowed to impersonate service accounts in their namespace. The **sudoers** role allows a user to impersonate **system:admin**, which in turn has cluster administrator permissions. This grants some protection against typos (but not security) for someone administering the cluster. For example, **oc delete nodes --all** would be forbidden, but **oc delete nodes --all --as=system:admin** would be allowed. You can add a user to that group using **oadm policy add-cluster-role-to-user sudoer <username>**.

4.5.4. OAuth

The OpenShift Container Platform 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.

4.5.4.1. 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 Container Platform 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.
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 1
secret: "..." 2
redirectURIs:
 - "http://www.example.com/" 3
grantMethod: prompt 4
')
```

1

The **name** of the OAuth client is used as the**client_id** parameter when making
requests to ***master*/oauth/authorize** and ***master*/oauth/token**.

2

The **secret** is used as the**client_secret** parameter when making requests to
***master*/oauth/token**.

3

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 URIs in **redirectURIs**.

4

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.

4.5.4.2. 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.

4.5.4.2.1. 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": ...,      1
     "name": ...,      2
     "group": ...      3
   }
}
```

1

       kind refers to the type of the object being referenced. Currently, only **route** is supported.

2

       name refers to the name of the object. The object must be in the same namespace as the service account.

3

       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"
```

4.5.4.3. 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 Container Platform 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 Container Platform 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 Container Platform 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.

4.5.4.4. OAuth Server Metadata

Applications running in OpenShift Container Platform 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 Container Platform 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.

4.5.4.5. 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

4.6. AUTHORIZATION

4.6.1. Overview

Authorization policies determine whether a user is allowed to perform a given action within a project. This allows platform administrators to use the cluster policy to control who has various access levels to the OpenShift Container Platform platform itself and all projects. It also allows developers to use local policy 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 *bound* to, multiple roles at the same time.
Binding s	Associations between users and/or groups with a role.

Cluster administrators can visualize rules, roles, and bindings using the CLI. For example, consider the following excerpt from viewing a policy, showing rule sets for the **admin** and **basic-user** default roles:

```
admin    Verbs       Resources                  Resource Names Extension
    [create delete get list update watch] [projects
resourcegroup:exposedkube resourcegroup:exposedopenshift
resourcegroup:granter secrets]     []
    [get list watch]   [resourcegroup:allkube resourcegroup:allkube-
status resourcegroup:allopenshift-status resourcegroup:policy]   []
basic-user  Verbs        Resources                 Resource Names Extension
    [get]      [users]                 [~]
    [list]      [projectrequests]          []
    [list]      [projects]            []
    [create]    [subjectaccessreviews]          []
IsPersonalSubjectAccessReview
```

The following excerpt from viewing policy bindings shows the above roles bound to various users and groups:

```
RoleBinding[admins]:
    Role: admin
    Users: [alice system:admin]
    Groups: []
RoleBinding[basic-user]:
    Role: basic-user
    Users: [joe]
    Groups: [devel]
```

The relationships between the the policy roles, policy bindings, users, and developers are illustrated below.
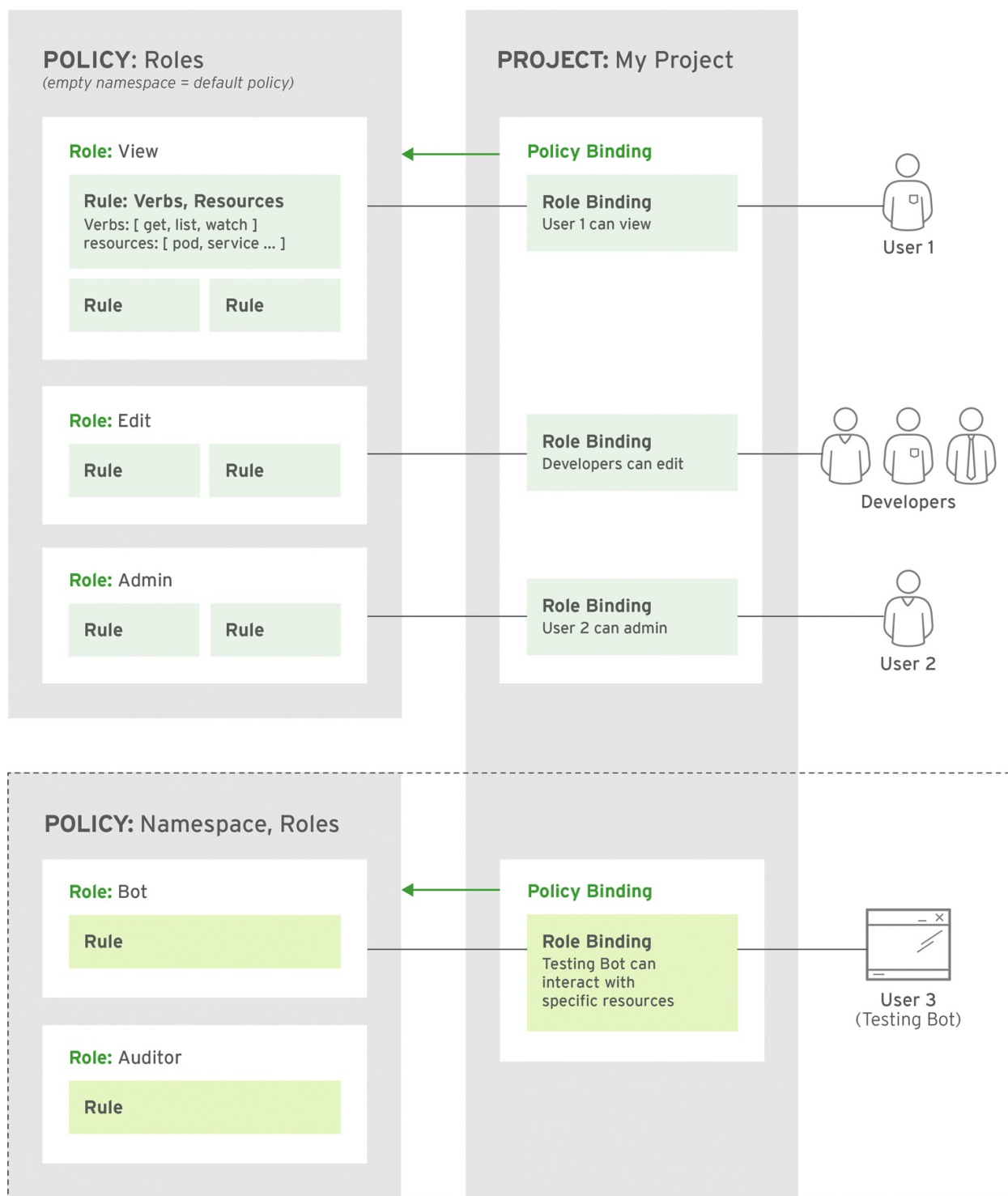
4.6.2. Evaluating Authorization

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

Identity	In the context of authorization, both the user name and list of groups the user belongs to.
Action	The action being performed. In most cases, this consists of:

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 Container Platform 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.

4.6.3. Cluster Policy and Local Policy

There are two levels of authorization policy:

Cluster policy	Roles and bindings that are applicable across all projects. Roles that exist in the cluster policy are considered *cluster roles*. Cluster bindings can only reference cluster roles.
Local policy	Roles and bindings that are scoped to a given project. Roles that exist only in a local policy are considered *local roles*. Local bindings can reference both cluster and local roles.

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

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

1. Cluster-wide "allow" rules are checked.

2. Locally-bound "allow" rules are checked.

3. Deny by default.

4.6.4. Roles

Roles are collections of policy rules, which are sets of permitted verbs that can be performed on a set of resources. OpenShift Container Platform includes a set of default roles that can be added to users and groups in the cluster policy or in a local policy.

Default 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 role creation and quota. If the **cluster-admin** wants to allow an **admin** to modify roles, the **cluster-admin** must create a project-scoped `Policy` object using JSON.
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 granted to a user within a local policy, they have full control over quota and roles and every action on every resource in the project.
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.

Cluster administrators can visualize these roles, including a matrix of the verbs and resources each are associated using the CLI to view the cluster roles. Additional **system:** roles are listed as well, which are used for various OpenShift Container Platform system and component operations.

By default in a local policy, only the binding for the **admin** role is immediately listed when using the CLI to view local bindings. However, if other default roles are added to users and groups within a local policy, they become listed in the CLI output, as well.

If you find that these roles do not suit you, a **cluster-admin** user can create a **policyBinding** object named *<projectname>*:**default** with the CLI using a JSON file. This allows the project **admin** to bind users to roles that are defined only in the *<projectname>* local policy.

Important

The **cluster-** role assigned by the project administrator is limited in a project. It is not the same **cluster-** role granted by 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.

Learn how to create a local role for a project

4.6.4.1. Updating Cluster Roles

After any OpenShift Container Platform cluster upgrade, the recommended default roles may have been updated. See Updating Policy Definitions for instructions on getting to the new recommendations using:

```
$ oc adm policy reconcile-cluster-roles
```

4.6.5. Security Context Constraints

In addition to authorization policies that control what a user can do, OpenShift Container Platform 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 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]
```

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: null
apiVersion: v1
defaultAddCapabilities: null
fsGroup: ❶
  type: RunAsAny
groups: ❷
- 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: null
runAsUser: ❸
  type: RunAsAny
seLinuxContext: ❹
  type: RunAsAny
supplementalGroups: ❺
  type: RunAsAny
users: ❻
- system:serviceaccount:default:registry
- system:serviceaccount:default:router
- system:serviceaccount:openshift-infra:build-controller
volumes:
- '*'
```

❶

The **FSGroup** strategy which dictates the allowable values for the Security
Context

❷

The groups that have access to this SCC

❸

The run as user strategy type which dictates the allowable values for the
Security Context

4

The SELinux context strategy type which dictates the allowable values for the Security Context

5

The supplemental groups strategy which dictates the allowable supplemental groups for the Security Context

6

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.

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.

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: » A mechanism to generate the value, and » A mechanism to ensure that a specified value falls into the set of allowable values.

4.6.5.1. SCC Strategies

4.6.5.1.1. 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.

4.6.5.1.2. 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.

4.6.5.1.3. 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.

4.6.5.1.4. 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.

4.6.5.2. 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**

» **flocker**

» **flexVolume**

» **hostPath**

» **emptyDir**

» **gcePersistentDisk**

» **awsElasticBlockStore**

» **gitRepo**

» **secret**

» **nfs**

» **iscsi**

» **glusterfs**

» **persistentVolumeClaim**

» **rbd**

» **cinder**

» **cephFS**

» **downwardAPI**

» **fc**

» **configMap**

» *****

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

> **Note**
>
> * is a special value to allow the use of all volume types.

> **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**.

4.6.5.3. 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.

Refer to the seccomp documentation for more information about configuring and using custom profiles.

4.6.5.4. 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.

4.6.5.4.1. 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.

4.6.5.4.2. 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.

4.6.6. Determining What You Can Do as an Authenticated User

From within your OpenShift Container Platform project, you can determine what verbs you can perform against all namespace-scoped resources (including third-party resources). Run:

```
$ oc policy can-i --list --loglevel=8
```

The output will help you to determine what API request to make to gather the information.

To receive information back in a user-readable format, run:

```
$ oc policy can-i --list
```

The output will provide a full list.

To determine if you can perform specific verbs, run:

```
$ oc policy can-i <verb> <resource>
```

User scopes can provide more information about a given scope. For example:

```
$ oc policy can-i <verb> <resource> --scopes=user:info
```

4.7. PERSISTENT STORAGE

4.7.1. Overview

Managing storage is a distinct problem from managing compute resources. OpenShift Container Platform 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 Container Platform 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).

4.7.2. 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.

4.7.2.1. Provisioning

A cluster administrator creates some number of PVs. They carry the details of the real storage that is available for use by cluster users. They exist in the API and are available for consumption.

4.7.2.2. Binding

A user creates a **PersistentVolumeClaim** with a specific amount of storage requested and with certain access modes. A control loop in the master watches for new PVCs, finds a matching PV (if possible), and binds them together. The user will always get at least what they asked for, but the volume may be in excess of what was requested. To minimize the excess, OpenShift Container Platform binds to the smallest PV that matches all other criteria.

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

4.7.2.3. 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.

4.7.2.4. 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.

4.7.2.5. Reclaiming

The reclaim policy of a **PersistentVolume** tells the cluster what to do with the volume after it is released. Currently, volumes can either be *retained* or *recycled*.

Retention allows for manual reclamation of the resource. For those volume plug-ins that support it, recycling performs a basic scrub on the volume (e.g., **rm -rf /<volume>/***) and makes it available again for a new claim.

4.7.3. Persistent Volumes

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

Example 4.1. 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
```

4.7.3.1. Types of Persistent Volumes

OpenShift Container Platform 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

4.7.3.2. Capacity

Generally, a PV will have a specific storage capacity. This is set using the PV's **capacity** attribute. See the Kubernetes Resource Model to understand the units expected by **capacity**.

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

4.7.3.3. 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.

Access Mode	CLI Abbreviation	Description
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 4.2. Supported Access Modes for Persistent Volumes

Volume Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
AWS EBS	X	-	-
Azure Disk	X	-	-
Ceph RBD	X	X	-
Fiber Channel	X	X	-
GCE Persistent Disk	X	-	-
GlusterFS	X	X	X

Volume Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
HostPath	X	-	-
iSCSI	X	X	-
NFS	X	X	X
Openstack Cinder	X	-	-

> **Note**
>
> ≫ If pods rely on AWS EBS, GCE Persistent Disks, or Openstack Cinder PVs, use a recreate deployment strategy
>
> ≫ Azure Disk does not support dynamic provisioning.

4.7.3.4. Recycling Policy

The current recycling policies are:

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

> **Note**
>
> Currently, only NFS and HostPath support the 'Recycle' recycling policy.

4.7.3.5. Phase

A 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.
Failed	The volume has failed its automatic reclamation.

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

4.7.4. Persistent Volume Claims

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

Example 4.2. Persistent Volume Claim Object Definition

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

4.7.4.1. Access Modes

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

4.7.4.2. 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.

4.7.4.3. 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
```

4.8. REMOTE COMMANDS

4.8.1. Overview

OpenShift Container Platform takes advantage of a feature built into Kubernetes to support executing commands in containers. This is implemented using HTTP along with a multiplexed streaming protocol such as **SPDY** or **HTTP/2**.

Developers can use the CLI to execute remote commands in containers.

4.8.2. Server Operation

The Kubelet handles remote execution requests from clients. Upon receiving a request, it upgrades the response, evaluates the request headers to determine what streams (**stdin**, **stdout**, and/or **stderr**) to expect to receive, and waits for the client to create the streams.

After the Kubelet has received all the streams, it executes the command in the container, copying between the streams and the command's stdin, stdout, and stderr, as appropriate. When the command terminates, the Kubelet closes the upgraded connection, as well as the underlying one.

Architecturally, there are options for running a command in a container. The supported implementation currently in OpenShift Container Platform invokes **nsenter** directly on the node host to enter the container's namespaces prior to executing the command. However, custom implementations could include using **docker exec**, or running a "helper" container that then runs **nsenter** so that **nsenter** is not a required binary that must be installed on the host.

4.9. PORT FORWARDING

4.9.1. Overview

OpenShift Container Platform takes advantage of a feature built into Kubernetes to support port forwarding to pods. This is implemented using HTTP along with a multiplexed streaming protocol such as **SPDY** or **HTTP/2**.

Developers can use the CLI to port forward to a pod. The CLI listens on each local port specified by the user, forwarding via the described protocol.

4.9.2. Server Operation

The Kubelet handles port forward requests from clients. Upon receiving a request, it upgrades the response and waits for the client to create port forwarding streams. When it receives a new stream, it copies data between the stream and the pod's port.

Architecturally, there are options for forwarding to a pod's port. The supported implementation currently in OpenShift Container Platform invokes **nsenter** directly on the node host to enter the pod's network namespace, then invokes **socat** to copy data between the stream and the pod's port. However, a custom implementation could include running a "helper" pod that then runs **nsenter** and **socat**, so that those binaries are not required to be installed on the host.

4.10. SOURCE CONTROL MANAGEMENT

OpenShift Container Platform 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 Container Platform 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 Container Platform. You can create a **BuildConfig** manually or let OpenShift Container Platform create it automatically by inspecting your repository.

> Triggering a build upon repository changes.

4.11. ADMISSION CONTROLLERS

4.11.1. 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 Container Platform master has a default list of plug-ins that are enabled by default for each type of resource (Kubernetes and OpenShift Container Platform). 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 Container Platform master.

4.11.2. General Admission Rules

Starting in 3.3, OpenShift Container Platform uses a single admission chain for Kubernetes and OpenShift Container Platform resources. This changed from 3.2, 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 3.3, 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.

4.11.3. Customizable Admission Plug-ins

Cluster administrators can configure some admission control plug-ins to control certain
behavior, such as:

- Limiting Number of Self-Provisioned Projects Per User

- Configuring Global Build Defaults and Overrides

- Controlling Pod Placement

4.11.4. Admission Controllers Using Containers

Admission controllers using containers also support init containers.

4.12. OTHER API OBJECTS

4.12.1. 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.

See the Kubernetes documentation for more information.

4.12.2. 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.

See Cluster Administration and Kubernetes documentation for more information on
ResourceQuota.

4.12.3. 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 and Kubernetes documentation for more information.

4.12.4. 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.

4.12.5. PersistentVolume

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

See the Kubernetes documentation for more information.

4.12.6. 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.

See the Kubernetes documentation for more information.

4.12.7. OAuth Objects

4.12.7.1. 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

Example 4.3. OAuthClient Object Definition

```
kind: "OAuthClient"
apiVersion: "v1"
metadata:
  name: "openshift-web-console"  1
  selflink: "/osapi/v1/oAuthClients/openshift-web-console"
  resourceVersion: "1"
  creationTimestamp: "2015-01-01T01:01:01Z"
respondWithChallenges: false  2
secret: "45e27750-a8aa-11e4-b2ea-3c970e4b7ffe"  3
redirectURIs:
  - "https://localhost:8443"  4
```

1

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

2

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

3

The value in the **secret** parameter is used as the **client_secret** parameter in an authorization code flow.

4

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**.

4.12.7.2. 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.

Example 4.4. 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"
userUID: "9311ac33-0fde-11e5-97a1-3c970e4b7ffe"
scopes: []
----
```

4.12.7.3. 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.

Example 4.5. 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
userUID: "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

userUID represents the UID of the User this token allows obtaining an OAuthAccessToken for.

4.12.7.4. 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.

Example 4.6. OAuthAccessToken Object Definition

```
kind: "OAuthAccessToken"
apiVersion: "v1"
metadata:
  name: "ODliOGE5ZmMtYzczYi00Nzk1LTg4MGEtNzQyZmUxZmUwY2Vh"  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
userUID: "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

userUID represents the User this token allows authentication as.

7

authorizeToken is the name of the OAuthAuthorizationToken used to obtain this token, if any.

4.12.8. User Objects

4.12.8.1. Identity

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

OpenShift Container Platform 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**.

Example 4.7. 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.

4.12.8.2. 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.

Example 4.8. 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.
```

4.12.8.3. 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**.

Example 4.9. UserIdentityMapping Object Definition

```
kind: "UserIdentityMapping"
apiVersion: "v1"
metadata:
  name: "anypassword:bob"        1
  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"

<1> `*UserIdentityMapping*` name matches the mapped `*Identity*` name
```

4.12.8.4. Group

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

Example 4.10. `Group` Object Definition

```
kind: "Group"
apiVersion: "v1"
metadata:
  name: "developers"   1
  creationTimestamp: "2015-01-01T01:01:01-00:00"
users:
  - "bob"   2
```

1   1   1

> **name** is the group name used when adding roles to a group.

2   2

> The values in **users** are the names of User objects that are members of this group.

---

[1] After this point, device names refer to devices on container B's host.

CHAPTER 5. REVISION HISTORY: ARCHITECTURE

5.1. WED MAY 31 2017

Affected Topic	Description of Change
Infrastructure Components → Container Registry	Updated to use OpenShift Container Registry name.

5.2. THU MAY 25 2017

Affected Topic	Description of Change
Core Concepts → Containers and Images	Added the Image Version Tag Policy section.

5.3. TUE MAY 02 2017

Affected Topic	Description of Change
Additional Concepts → F5 BIG-IP® Router Plug-in	Removed information about integration with HAProxy router.
Additional Concepts → F5 BIG-IP® Router Plug-in	F5 BIG-IP® Router Plug-in is now its own topic.

5.4. WED APR 12 2017

Affected Topic	Description of Change
Core Concepts → Routes	Updated router customization details.

5.5. MON MAR 20 2017

Affected Topic	Description of Change
Core Concepts → Routes	Added the Denying or Allowing Certain Domains in Routes section.
Additional Concepts → Flannel	Updated the topic with information on OpenStack installs, running Flannel in **host-gw** mode, and a diagram with route examples.

5.6. MON MAR 06 2017

Affected Topic	Description of Change
Core Concepts → Routes	Clarified that `hostsubnet` requires a specific annotation for multi-tenancy to work.

5.7. THU FEB 16 2017

Affected Topic	Description of Change
Infrastructure Components → Web Console	Removed the Technical Preview note from the JVM Console section.
Core Concepts → Routes	Edited out the **ROUTE_LABELS** environment variable to allighn with upstream.
	Added descriptions for **ROUTER_ALLOWED_DOMAINS** and **ROUTER_DENIED_DOMAINS** in the Router Configuration Parameters table.

5.8. MON FEB 06 2017

Affected Topic	Description of Change
Core Concepts → Routes	Added a new F5 Native Integration section.
Additional Concepts → OpenShift SDN	Updated OpenShift SDN information.

5.9. WED JAN 25 2017

Affected Topic	Description of Change
Core Concepts → Pods and Services	Updated Ingress CIDR references to the new default.

5.10. WED JAN 18 2017

OpenShift Container Platform 3.4 initial release.

Affected Topic	Description of Change
Core Concepts → Containers and Images	New section about init containers.
Core Concepts → Builds and Image Streams	Added information on how the first time a project defines a build configuration using a Pipeline strategy, OpenShift Container Platform instantiates a Jenkins server to execute the pipeline.
	Added new Image Stream Image and Image Stream Tag sections.
	Added a link for more details on how the Jenkins server is deployed.
Core Concepts → Routes	Added descriptions of all route annotations.

Affected Topic	Description of Change
	Added two new parameters under Configuration Parameters in the HAProxy Template Router section.
	Added a new Creating Routes Specifying a Wildcard Subdomain Policy section.
Additional Concepts → Authentication	Added new Service Accounts as OAuth Clients and OAuth Server Metadata sections.