# OpenShift Enterprise 3.0 Architecture

OpenShift Enterprise 3.0 Architecture Information

Red Hat OpenShift Documentation Team

# OpenShift Enterprise 3.0 Architecture

OpenShift Enterprise 3.0 Architecture Information

## Abstract

Learn the architecture of OpenShift Enterprise 3.0 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 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.

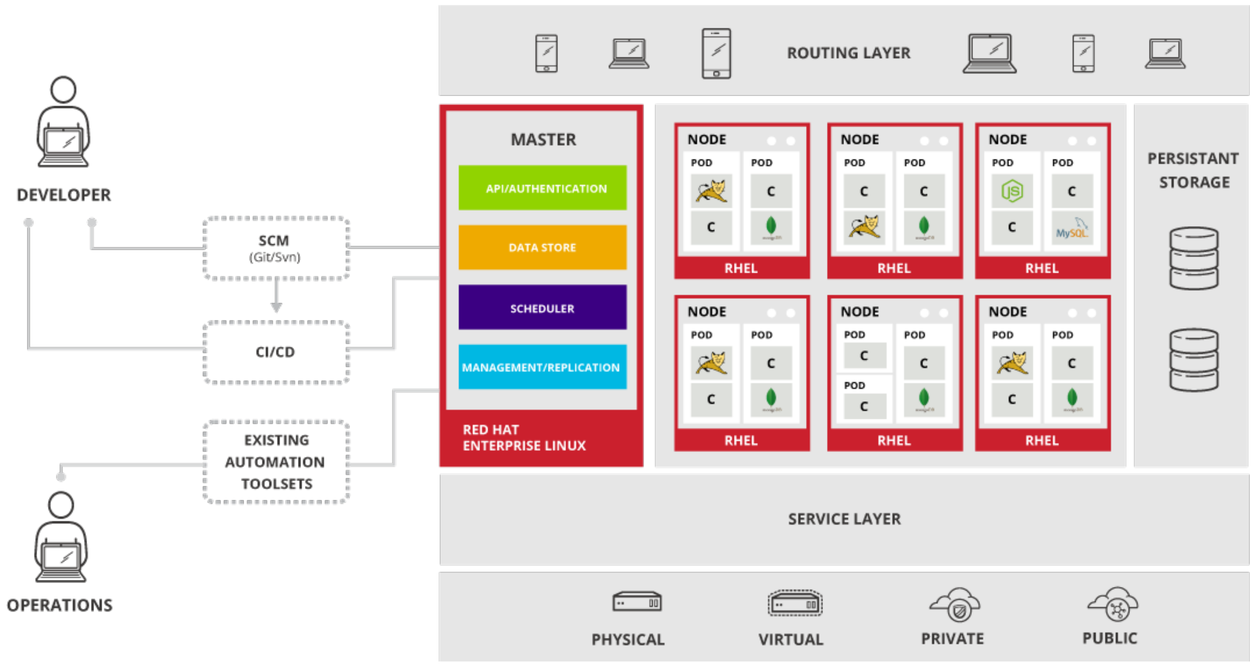
## 1.1. WHAT ARE THE LAYERS?

Docker provides the abstraction for packaging and creating Linux-based, lightweight containers. Kubernetes provides the cluster management and orchestrates Docker containers on multiple hosts.

OpenShift 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 Architecture Overview**



## 1.2. WHAT IS THE OPENSHIFT ARCHITECTURE?

OpenShift has a microservices-based architecture of smaller, decoupled units that work together. It can run on top of (or alongside) 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 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 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 SECURED?

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

A Kubernetes cluster consists of one or more masters and a set of nodes.

### 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. |
| **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. |
| Pacemaker | Optional, used when configuring highly-available masters.<br><br>Pacemaker is the core technology of the High Availability Add-on for Red Hat Enterprise Linux, providing consensus, fencing, and service management. It can be run on all master hosts to ensure that all active-passive components have one instance running. |

| Component | Description |
|---|---|
| Virtual IP | Optional, used when configuring highly-available masters.<br><br>The virtual IP (VIP) is the single point of contact, but not a single point of failure, for all OpenShift clients that:<br><br>- cannot be configured with all master service endpoints, or<br>- do not know how to load balance across multiple masters nor retry failed master service connections.<br><br>There is one VIP and it is managed by Pacemaker. |

#### 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 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 Pacemaker as the management technology, which Red Hat recommends. However, you can take the concepts and apply them towards your existing high availability solutions.

**Note**

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

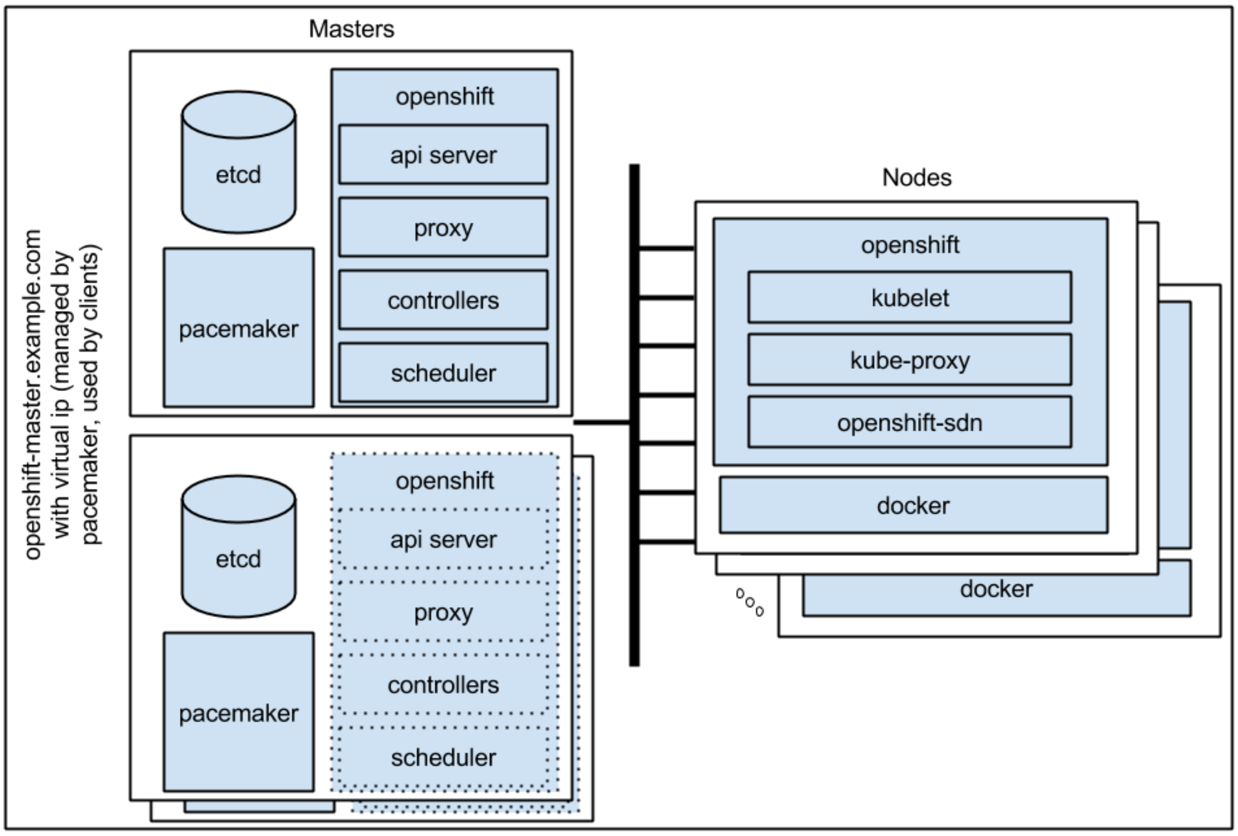
When using Pacemaker, master components have the following availability:

**Table 2.2. Availability Matrix**

| Role | Style | Notes |
|---|---|---|
| **etcd** | Active-active | Fully redundant deployment with load balancing |

| Role | Style | Notes |
|---|---|---|
| Master service | Active-passive | One active at a time, managed by Pacemaker |
| Pacemaker | Active-active | Fully redundant deployment |
| Virtual IP | Active-passive | One active at a time, managed by Pacemaker |

**Figure 2.1. Highly-available Masters Using Pacemaker**



## 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 Docker, a kubelet, and a service proxy.

OpenShift 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 instance using the CLI. To define full configuration and security options when launching node servers, use dedicated node configuration files.

#### 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. IMAGE REGISTRY

### 2.2.1. Overview

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

### 2.2.2. Integrated OpenShift Registry

OpenShift provides an integrated Docker registry that adds the ability to provision new image repositories on the fly. This allows users to automatically have a place for their builds to push the resulting images.

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

### 2.2.3. Third Party Registries

OpenShift 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 registry. In this situation OpenShift 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.

To create an image stream from an external registry, set the **`spec.dockerImageRepository`** field appropriately. For example:

```
{
  "apiVersion": "v1",
  "kind": "ImageStream",
  "metadata": {
    "name": "ruby"
  },
  "spec": {
    "dockerImageRepository": "openshift/ruby-20-centos7"
  }
}
```

After OpenShift synchronizes the tag and image metadata, it looks something like this:

```
{
  "kind": "ImageStream",
  "apiVersion": "v1",
  "metadata": {
    "name": "ruby",
    "namespace": "default",
    "selfLink": "/osapi/v1/namespaces/default/imagestreams/ruby",
    "uid": "9990ea5f-f35a-11e4-937e-001c422dcd49",
    "resourceVersion": "53",
    "creationTimestamp": "2015-05-05T19:11:57Z",
    "annotations": {
      "openshift.io/image.dockerRepositoryCheck": "2015-05-
05T19:12:00Z"
    }
  },
  "spec": {
    "dockerImageRepository": "openshift/ruby-20-centos7"
  },
  "status": {
    "dockerImageRepository": "openshift/ruby-20-centos7",
    "tags": [
      {
        "tag": "latest",
        "items": [
          {
            "created": "2015-05-05T19:11:58Z",
            "dockerImageReference": "openshift/ruby-20-
centos7:latest",
            "image":
"94439378e4546d72ef221c47fe2ac30065bcc3a98c25bc51bed77ec00efabb95"
          }
        ]
      },
      {
        "tag": "v0.4",
        "items": [
          {
            "created": "2015-05-05T19:11:59Z",
            "dockerImageReference": "openshift/ruby-20-centos7:v0.4",
            "image":
```

```
"c7dbf059225847a7bfb4f40bc335ad7e70defc913de1a28aabea3a2072844a3f"
              }
            ]
          }
        ]
      }
    }
```

**Note**

Querying external registries to synchronize tag and image metadata is not currently an automated process. To resynchronize manually, run **`oc import-image <stream>`**. Within a short amount of time, OpenShift will communicate with the external registry to get up to date information about the Docker image repository associated with the image stream.

#### 2.2.3.1. Authentication

OpenShift 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 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.

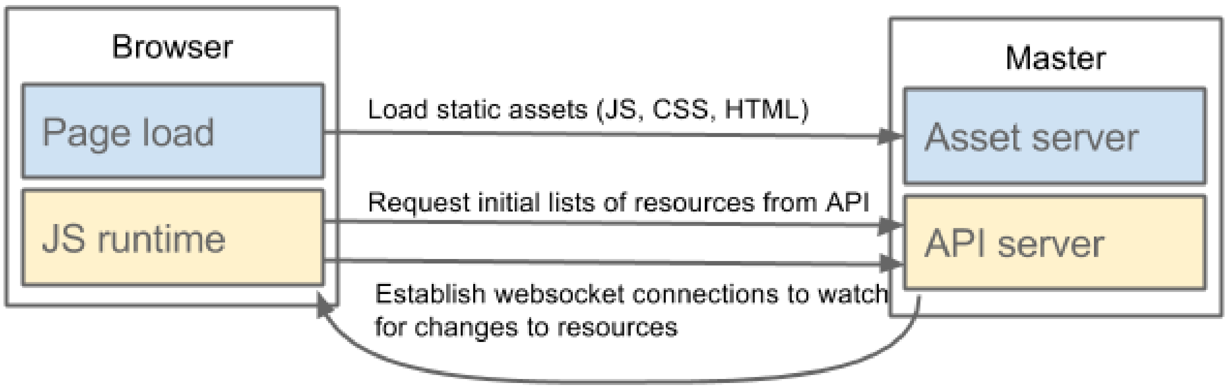
The web console is started as part of the master. All static assets required to run the web console are served from the **`openshift`** binary. Administrators can also customize the web console using extensions, which let you run scripts and load custom stylesheets when the web console loads. You can 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 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.

**Note**

JavaScript must be enabled to use the web console. For the best experience, use a web browser that supports WebSockets.

**Figure 2.2. Web Console Request Architecture**


Browser
Page load
JS runtime
Load static assets (JS, CSS, HTML)
Request initial lists of resources from API
Establish websocket connections to watch for changes to resources
Master
Asset server
API server

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. Browser Requirements

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

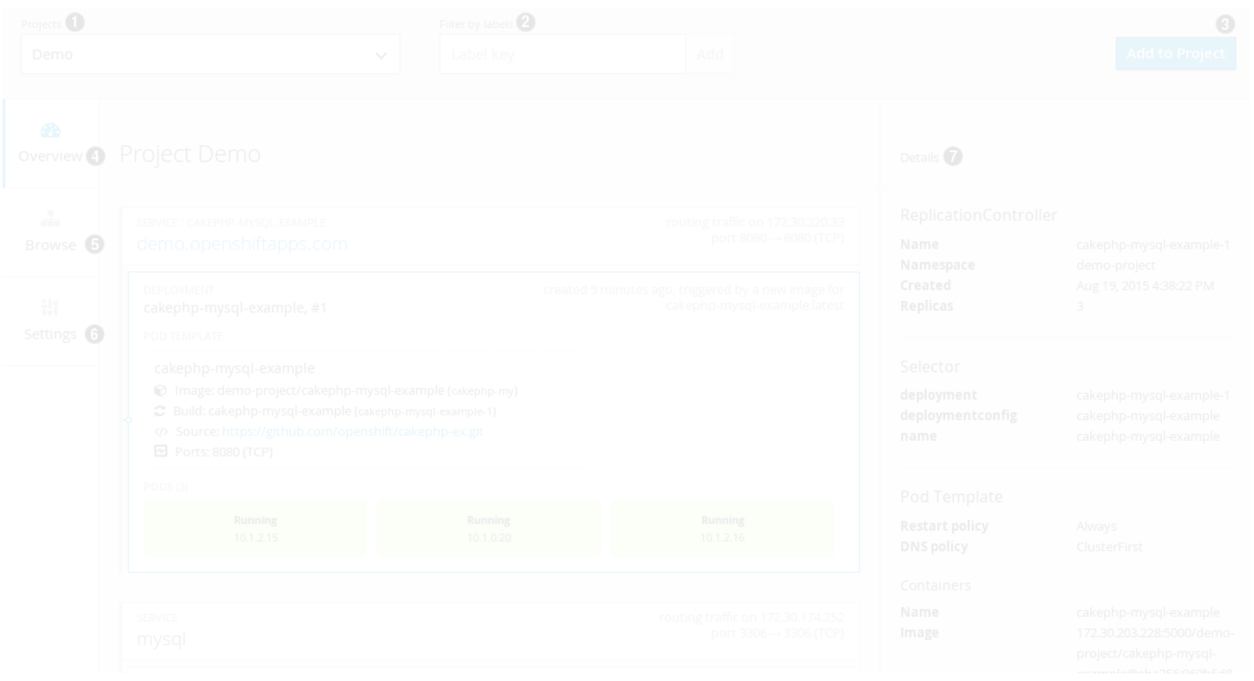
**Table 2.3. Browser Requirements**

| 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 |

### 2.3.3. Project Overviews

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

**Figure 2.3. Web Console Project Overview**



SERVICE : CAKEPHP-MYSQL-EXAMPLE
demo.openshiftapps.com
routing traffic on 172.30.220.33
port 8080 → 8080 (TCP)

DEPLOYMENT
cakephp-mysql-example, #1
created 9 minutes ago, triggered by a new image for cakephp-mysql-example:latest

POD TEMPLATE
cakephp-mysql-example
Image: demo-project/cakephp-mysql-example (cakephp-my)
Build: cakephp-mysql-example (cakephp-mysql-example-1)
Source: https://github.com/openshift/cakephp-ex.git
Ports: 8080 (TCP)

PODS (3)
Running 10.1.2.15
Running 10.1.0.20
Running 10.1.2.16

SERVICE
mysql
routing traffic on 172.30.174.252
port 3306 → 3306 (TCP)

ReplicationController

| | |
|---|---|
| Name | cakephp-mysql-example-1 |
| Namespace | demo-project |
| Created | Aug 19, 2015 4:38:22 PM |
| Replicas | 3 |

Selector

| | |
|---|---|
| deployment | cakephp-mysql-example-1 |
| deploymentconfig | cakephp-mysql-example |
| name | cakephp-mysql-example |

Pod Template

| | |
|---|---|
| Restart policy | Always |
| DNS policy | ClusterFirst |

Containers

| | |
|---|---|
| Name | cakephp-mysql-example |
| Image | 172.30.203.228:5000/demo-project/cakephp-mysql- |

The project selector allows you to switch between projects you have access to.

Filter the contents of a project page by using the labels of a resource.

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.

The **Browse** tab explores the different objects types within your project: Builds, Deployments, Image Streams, Pods, and Services.

The **Settings** tab provides general information about your project, as well as the quota and resource limits that are set on your project.

When you click on one of your objects in the **Overview** page, the **Details** pane displays detailed information about that object. In this example, the **cakephp-mysql-example** deployment is selected, and the **Details** pane is displaying details on the related replication controller.

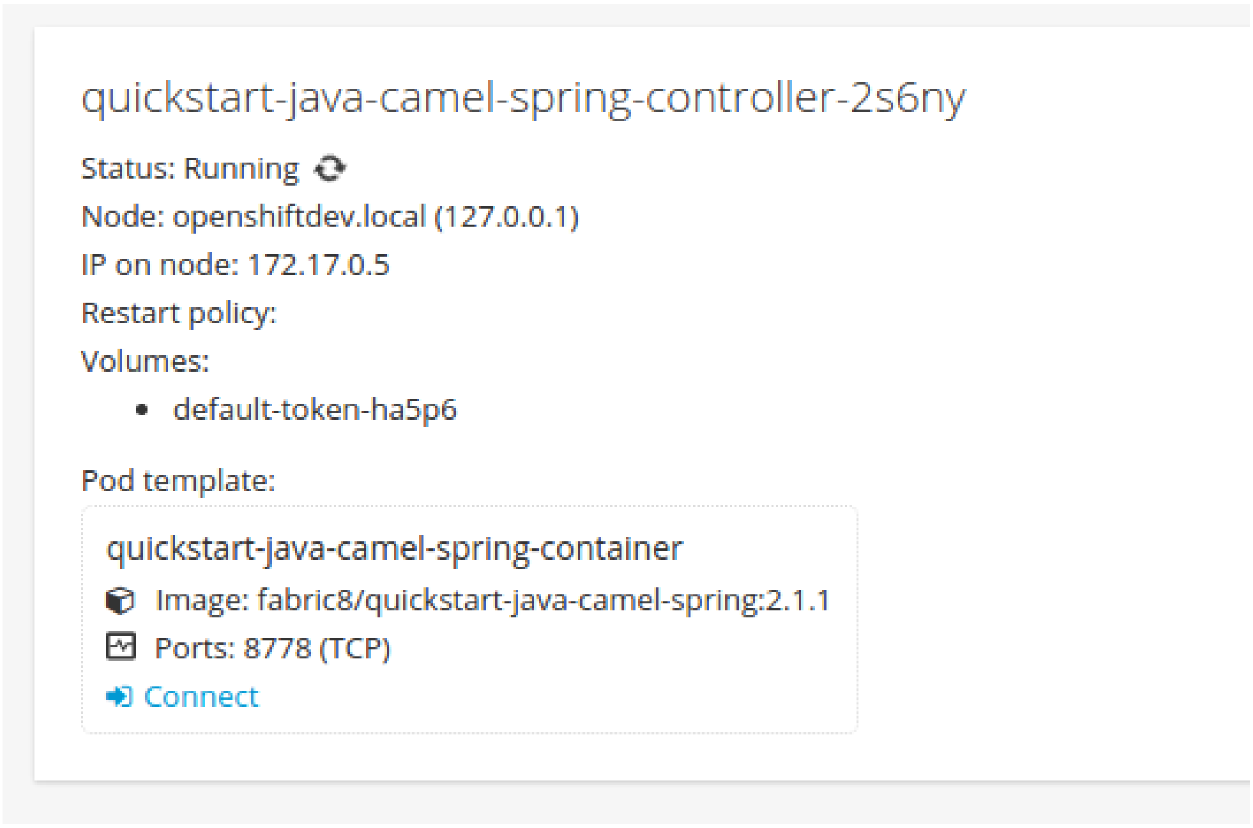
### 2.3.4. JVM Console

**Note**

This feature is currently in Technology Preview and not intended for production use.

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 2.4. Pod with a Link to the JVM Console**



quickstart-java-camel-spring-controller-2s6ny

Status: Running

Node: openshiftdev.local (127.0.0.1)

IP on node: 172.17.0.5

Restart policy:

Volumes:

- default-token-ha5p6

Pod template:

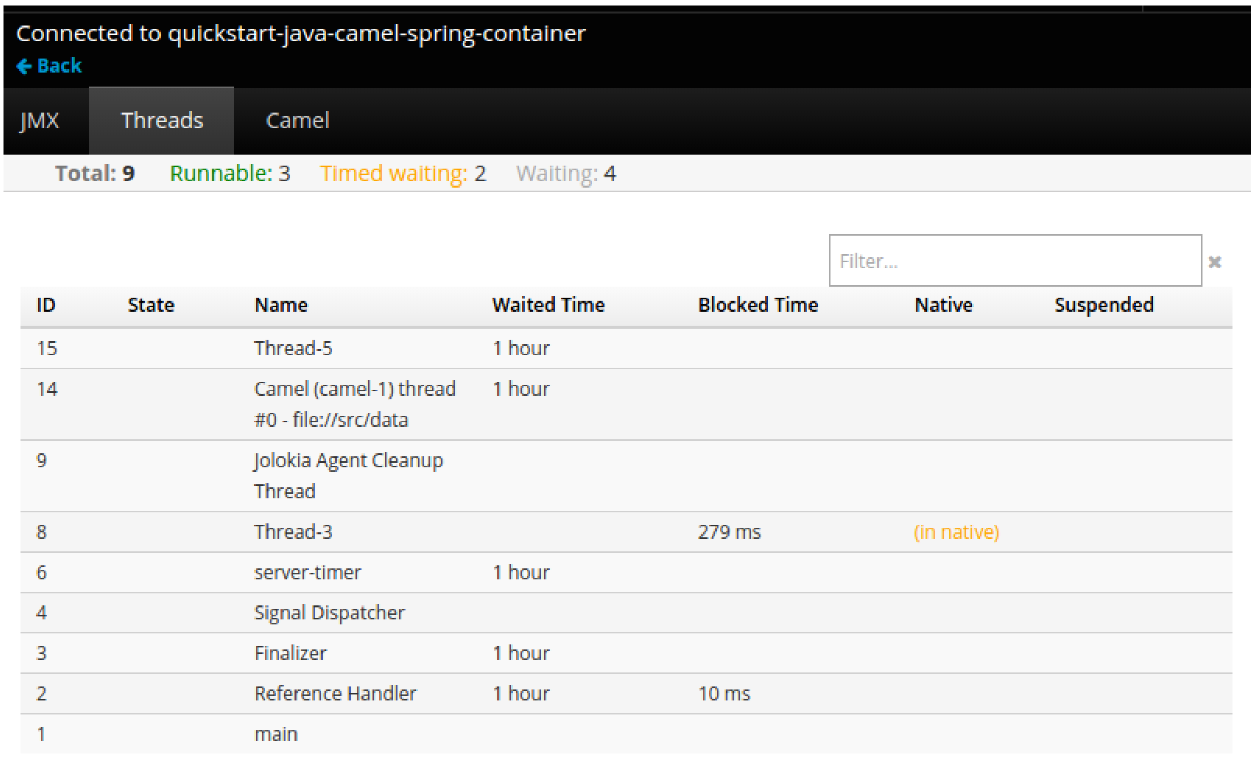
quickstart-java-camel-spring-container

Image: fabric8/quickstart-java-camel-spring:2.1.1

Ports: 8778 (TCP)

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

**Figure 2.5. JVM Console**


Connected to quickstart-java-camel-spring-container

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

| 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. Many of these objects come from Kubernetes, which is extended by OpenShift 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 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 and Kubernetes add the ability to orchestrate Docker containers across multi-host installations.

Though you do not directly interact with Docker tools when using OpenShift, understanding Docker’s capabilities and terminology is important for understanding its role in OpenShift and how your applications function inside of containers. Docker is available as part of RHEL 7, as well as CentOS and Fedora, so you can experiment with it separately from OpenShift. Refer to the article Get Started with Docker Formatted Container Images on Red Hat Systems for a guided introduction.

### 3.2.2. Docker Images

Docker containers are based on Docker images. A Docker image is a binary that includes all of the requirements for running a single Docker container, as well as metadata describing its needs and capabilities. You can think of it as a packaging technology. Docker 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 can provide redundancy and horizontal scaling for a service packaged into an image.

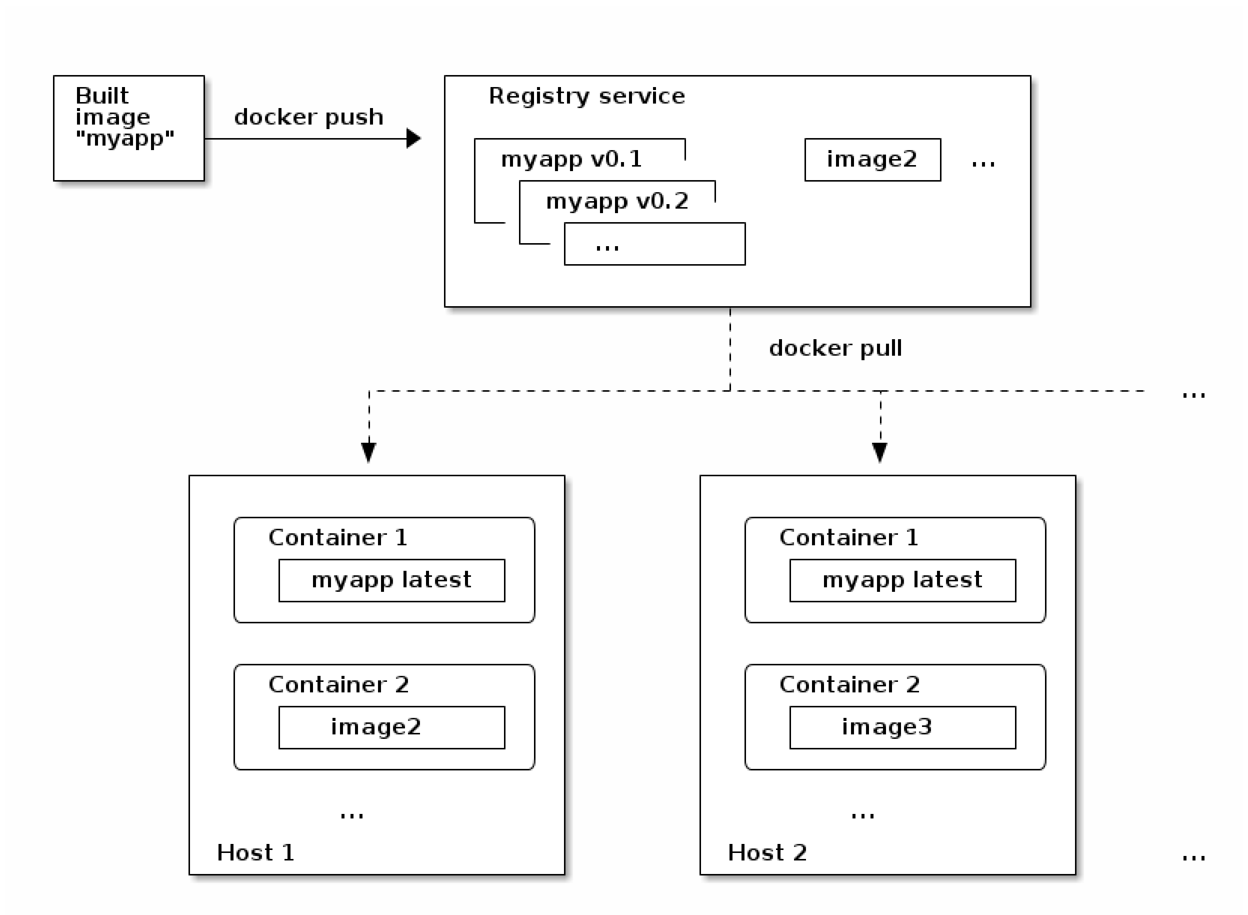
You can use Docker directly to build images, but OpenShift also supplies builders that assist with creating an image by adding your code or configuration to existing images.

Since 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`**). Rather than version numbers, Docker 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`**.

### 3.2.3. Docker Registries

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

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



## 3.3. PODS AND SERVICES

### 3.3.1. Pods

OpenShift 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 OpenShift v2 gears, with containers the rough equivalent of v2 cartridge instances. 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 treats pods as largely immutable; changes cannot be made to a pod definition while it is running. OpenShift 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.

Below is an example definition of a pod that provides a long-running service, which is actually a part of the OpenShift infrastructure: the private Docker 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 image.

6

The container can bind to ports which will be made available on the pod’s IP.

7

OpenShift 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 API.

9

Pods making requests against the OpenShift 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 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 internal address.

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
```

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.

Port on the backing pods to which the service forwards connections.

The Kubernetes documentation has more information on services.

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

## 3.4. PROJECTS AND USERS

### 3.4.1. Users

Interaction with OpenShift is associated with a user. An OpenShift 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 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. 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, 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 leverages Kubernetes by creating Docker containers from build images and pushing them to a Docker 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 build system provides extensible support for *build strategies* that are based on selectable types specified in the build API. There are three 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.

For a list of build commands, see the Developer's Guide.

For more information on how OpenShift leverages Docker for builds, see the upstream documentation.

#### 3.5.1.1. Docker Build

The Docker build strategy invokes the plain 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 images. It produces ready-to-run images by injecting application source into a Docker image and assembling a new Docker 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 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. |

#### 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.

The Custom builder image is a plain Docker image with embedded build process logic, such as building RPMs or building base Docker images. The openshift/origin-custom-docker-builder image is used by default.

### 3.5.2. Image Streams

An *image stream* can be used to automatically perform an action, such as updating a deployment, when a new image, such as a new version of the base image that is used in that deployment, is created.

An image stream comprises one or more Docker images identified by tags. It presents a single virtual view of related images, similar to a Docker image repository, and may contain images from any of the following:

1. Its own image repository in OpenShift's integrated Docker Registry
2. Other image streams
3. Docker image repositories from external registries

OpenShift components such as builds and deployments can watch an image stream to receive notifications when new images are added and react by performing a build or a deployment.

**Example 3.3. Image Stream Object Definition**

```
{
  "kind": "ImageStream",
  "apiVersion": "v1",
  "metadata": {
    "name": "origin-ruby-sample",
    "namespace": "p1",
    "selfLink": "/osapi/v1/namesapces/p1/imageStreams/origin-ruby-
sample",
    "uid": "480dfe73-f340-11e4-97b5-001c422dcd49",
    "resourceVersion": "293",
    "creationTimestamp": "2015-05-05T16:03:34Z",
    "labels": {
      "template": "application-template-stibuild"
    }
  },
  "spec": {},
  "status": {
    "dockerImageRepository": "172.30.30.129:5000/p1/origin-ruby-
sample",
    "tags": [
      {
        "tag": "latest",
        "items": [
          {
            "created": "2015-05-05T16:05:47Z",
            "dockerImageReference": "172.30.30.129:5000/p1/origin-
ruby-
sample@sha256:4d3a646b58685449179a0c61ad4baa19a8df8ba668e0f0704b9ad16f5
e16e642",
            "image":
"sha256:4d3a646b58685449179a0c61ad4baa19a8df8ba668e0f0704b9ad16f5e16e64
2"
          }
        ]
        }
      ]
    }
  }
```

### 3.5.2.1. Image Stream Mappings

When the integrated OpenShift Docker Registry receives a new image, it creates and sends an **`ImageStreamMapping`** to OpenShift. This informs OpenShift of the image's namespace, name, tag, and Docker metadata. OpenShift uses this information to create a new image (if it does not already exist) and to tag the image into the image stream. OpenShift stores complete metadata about each image (e.g., command, entrypoint, environment variables, etc.). Note that images in OpenShift are immutable. Also, note that the maximum name length is 63 characters.

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

**Example 3.4. Image Stream Mapping Object Definition**

```
{
  "kind": "ImageStreamMapping",
  "apiVersion": "v1",
  "metadata": {
    "name": "origin-ruby-sample",
    "namespace": "test"
  },
  "image": {
    "metadata": {
      "name":
"a2f15cc10423c165ca221f4a7beb1f2949fb0f5acbbc8e3a0250eb7d5593ae64"
    },
    "dockerImageReference": "172.30.17.3:5001/test/origin-ruby-
sample:a2f15cc10423c165ca221f4a7beb1f2949fb0f5acbbc8e3a0250eb7d5593ae64
",
    "dockerImageMetadata": {
      "kind": "DockerImage",
      "apiVersion": "1.0",
      "Id":
"a2f15cc10423c165ca221f4a7beb1f2949fb0f5acbbc8e3a0250eb7d5593ae64",
      "Parent":
"3bb14bfe4832874535814184c13e01527239633627cdc38f18fa186e73a6b62c",
      "Created": "2015-01-23T21:47:04Z",
      "Container":
"f81db8980c62d7650683326173a361c3b09f3bc41471918b6319f7df67943b54",
      "ContainerConfig": {
        "Hostname": "f81db8980c62",
        "User": "ruby",
        "AttachStdout": true,
        "ExposedPorts": {
          "9292/tcp": {}
        },
        "OpenStdin": true,
        "StdinOnce": true,
        "Env": [
          "OPENSHIFT_BUILD_NAME=4bf65438-a349-11e4-bead-001c42c44ee1",
          "OPENSHIFT_BUILD_NAMESPACE=test",
          "OPENSHIFT_BUILD_SOURCE=https://github.com/openshift/ruby-
hello-world",

"PATH=/opt/ruby/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin",

"STI_SCRIPTS_URL=https://raw.githubusercontent.com/openshift/sti-
ruby/master/2.0/.sti/bin",
          "APP_ROOT=.",
          "HOME=/opt/ruby"
        ],
        "Cmd": [
          "/bin/sh",
          "-c",
          "tar -C /tmp -xf - \u0026\u0026 /tmp/scripts/assemble"
        ],
        "Image": "openshift/ruby-20-centos7",
        "WorkingDir": "/opt/ruby/src"
      },
      "DockerVersion": "1.4.1-dev",
      "Config": {
        "User": "ruby",
        "ExposedPorts": {
          "9292/tcp": {}
        },
        "Env": [
          "OPENSHIFT_BUILD_NAME=4bf65438-a349-11e4-bead-001c42c44ee1",
          "OPENSHIFT_BUILD_NAMESPACE=test",
          "OPENSHIFT_BUILD_SOURCE=https://github.com/openshift/ruby-
hello-world",

"PATH=/opt/ruby/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin",

"STI_SCRIPTS_URL=https://raw.githubusercontent.com/openshift/sti-
ruby/master/2.0/.sti/bin",
          "APP_ROOT=.",
          "HOME=/opt/ruby"
        ],
        "Cmd": [
          "/tmp/scripts/run"
        ],
        "WorkingDir": "/opt/ruby/src"
      },
      "Architecture": "amd64",
      "Size": 11710004
    },
    "dockerImageMetadataVersion": "1.0"
  },
  "tag": "latest"
}
```

### 3.5.2.2. Referencing Images in Image Streams

An **`ImageStreamTag`** is used to reference or retrieve an image for a given image stream and tag. It uses the following convention for its name: **`<image stream name>:<tag>`**.

An **`ImageStreamImage`** is used to reference or retrieve an image for a given image stream and image name. It uses the following convention for its name: **`<image stream name>@<name>`**.

The sample image below is from the **ruby** image stream and was retrieved by asking for the **`ImageStreamImage`** with the name **ruby@371829c**:

**Example 3.5. Definition of an Image Object retrieved via ImageStreamImage**

```
{
    "kind": "ImageStreamImage",
    "apiVersion": "v1",
    "metadata": {
        "name": "ruby@371829c",
        "uid": "a48b40d7-18e2-11e5-9ba2-001c422dcd49",
        "resourceVersion": "1888",
        "creationTimestamp": "2015-06-22T13:29:00Z"
    },
    "image": {
        "metadata": {
            "name":
"371829c6d5cf05924db2ab21ed79dd0937986a817c7940b00cec40616e9b12eb",
            "uid": "a48b40d7-18e2-11e5-9ba2-001c422dcd49",
            "resourceVersion": "1888",
            "creationTimestamp": "2015-06-22T13:29:00Z"
        },
        "dockerImageReference": "openshift/ruby-20-centos7:latest",
        "dockerImageMetadata": {
            "kind": "DockerImage",
            "apiVersion": "1.0",
            "Id":
"371829c6d5cf05924db2ab21ed79dd0937986a817c7940b00cec40616e9b12eb",
            "Parent":
"8c7059377eaf86bc913e915f064c073ff45552e8921ceeb1a3b7cbf9215ecb66",
            "Created": "2015-06-20T23:02:23Z",
            "ContainerConfig": {},
            "DockerVersion": "1.6.0",
            "Author": "Jakub Hadvig \u003cjhadvig@redhat.com\u003e",
            "Config": {
                "User": "1001",
                "ExposedPorts": {
                    "8080/tcp": {}
                },
                "Env": [

"PATH=/opt/openshift/src/bin:/opt/openshift/bin:/usr/local/sti:/usr/loc
al/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
                    "STI_SCRIPTS_URL=image:///usr/local/sti",
                    "HOME=/opt/openshift/src",
                    "BASH_ENV=/opt/openshift/etc/scl_enable",
                    "ENV=/opt/openshift/etc/scl_enable",
                    "PROMPT_COMMAND=. /opt/openshift/etc/scl_enable",
                    "RUBY_VERSION=2.0"
                ],
                "Cmd": [
                    "usage"
                ],
                "Image":
"8c7059377eaf86bc913e915f064c073ff45552e8921ceeb1a3b7cbf9215ecb66",
                "WorkingDir": "/opt/openshift/src",
                "Labels": {
                    "io.openshift.s2i.scripts-url":
"image:///usr/local/sti",
                    "k8s.io/description": "Platform for building and
running Ruby 2.0 applications",
                    "k8s.io/display-name": "Ruby 2.0",
                    "openshift.io/expose-services": "8080:http",
                    "openshift.io/tags": "builder,ruby,ruby20"
                }
            },
            "Architecture": "amd64",
            "Size": 53950504
        },
        "dockerImageMetadataVersion": "1.0"
    }
}
```

#### 3.5.2.3. Image Pull Policy

Each container in a pod has a Docker image. Once you have created an image and pushed it to a registry, you can then refer to it in the pod.

When OpenShift creates containers, it uses the container's **`imagePullPolicy`** to determine if the image should be pulled prior to starting the container. There are three possible values for **`imagePullPolicy`**:

- **`Always`** - always pull the image.
- **`IfNotPresent`** - only pull the image if it does not already exist on the node.
- **`Never`** - never pull the image.

If a container's **`imagePullPolicy`** parameter is not specified, OpenShift sets it based on the image's tag:

1. If the tag is **latest**, OpenShift defaults **`imagePullPolicy`** to **`Always`**.
2. Otherwise, OpenShift defaults **`imagePullPolicy`** to **`IfNotPresent`**.

#### 3.5.2.4. Importing Tag and Image Metadata

An image stream can be configured to import tag and image metadata from an image repository in an external Docker image registry. See Image Registry for more details.

#### 3.5.2.5. Tag Tracking

An image stream can also be configured so that a tag "tracks" another one. For example, you can configure the **latest** tag to always refer to the current image for the tag "2.0":

```
{
  "kind": "ImageStream",
  "apiVersion": "v1",
  "metadata": {
    "name": "ruby"
  },
  "spec": {
    "tags": [
      {
        "name": "latest",
        "from": {
          "kind": "ImageStreamTag",
          "name": "2.0"
        }
      }
    ]
  }
}
```

```
$ oc tag ruby:latest ruby:2.0
```

### 3.5.2.6. Tag Removal

You can stop tracking a tag by removing it. For example, you can stop tracking the **latest** tag you set above:

```
$ oc tag -d ruby:latest
```

**Important**

The above command removes the tag from the image stream spec, but not from the image stream status. The image stream spec is user-defined, whereas the image stream status reflects the information the system has from the specification. To remove a tag completely from an image stream:

```
$ oc delete istag/ruby:latest
```

You can also do the same using the `oc tag` command:

**`oc tag ruby:latest ruby:2.0`**

### 3.5.2.7. Importing Images from Insecure Registries

An image stream can be configured to import tag and image metadata from an image repository that is signed with a self-signed certificate or from one using plain http instead of https. To do that, add **`openshift.io/image.insecureRepository`** annotation set to **`true`**. This setting will bypass certificate validation when connecting to registry.

```
kind: ImageStream
apiVersion: v1
metadata:
  name: ruby
  annotations:
    openshift.io/image.insecureRepository: "true"
  spec:
    dockerImageRepository: "my.repo.com:5000/myimage"
```

**Important**

The above definition will only affect importing tag and image metadata. For this image to be used in the cluster (be able to do **`docker pull`**) each node needs to have docker configured with **`--insecure-registry`** flag. See Installing Docker in Host Preparation for information on the topic.

## 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 replica controller acts to instantiate more up to the desired number. Likewise, if there are more running than desired, it deletes as many as necessary to match the number.

The definition of a replication controller consists mainly of:

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

The selector is just a set of labels that all of the pods managed by the replication controller should have. So that set of labels is included in the pod definition that the replication controller instantiates. This selector is used by the replication controller to determine how many instances of the pod are already running in order to adjust as needed.

It is not the job of the replication controller to 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.

Replication controllers are a core Kubernetes object, **`ReplicationController`**. The Kubernetes documentation has more details on replication controllers.

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

```
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. Deployments and Deployment Configurations

Building on replication controllers, OpenShift 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 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 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 route is a way to expose a service by giving it an externally-reachable hostname like **www.example.com**.

A defined route and the endpoints identified by its service can be consumed by a router to provide named connectivity that allows external clients to reach your applications. Each route consists of a route name (limited to 63 characters), service selector, and (optionally) security configuration.

### 3.7.2. Routers

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

OpenShift routers provide external host name mapping and load balancing to services 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 and 443. This is to allow external traffic to route to the host and subsequently through the router. Routers also assume that networking is configured such that it 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.

A router uses the service selector to find the service and the endpoints backing the service. Service-provided load balancing is bypassed and replaced with the router's own load balancing. Routers watch the cluster API and automatically update their own configuration according to any relevant changes in the API objects. Routers may be containerized or virtual. Custom routers can be deployed to communicate modifications of API objects to an external routing solution.

In order to reach a router in the first place, requests for host names must resolve via DNS to a router or set of routers. The suggested method is to define a cloud domain with a wildcard DNS entry pointing to a virtual IP backed by multiple router instances on designated nodes. Router VIP configuration is described in the Administration Guide. DNS for addresses outside the cloud domain would need to be configured individually. Other approaches may be feasible.

#### 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

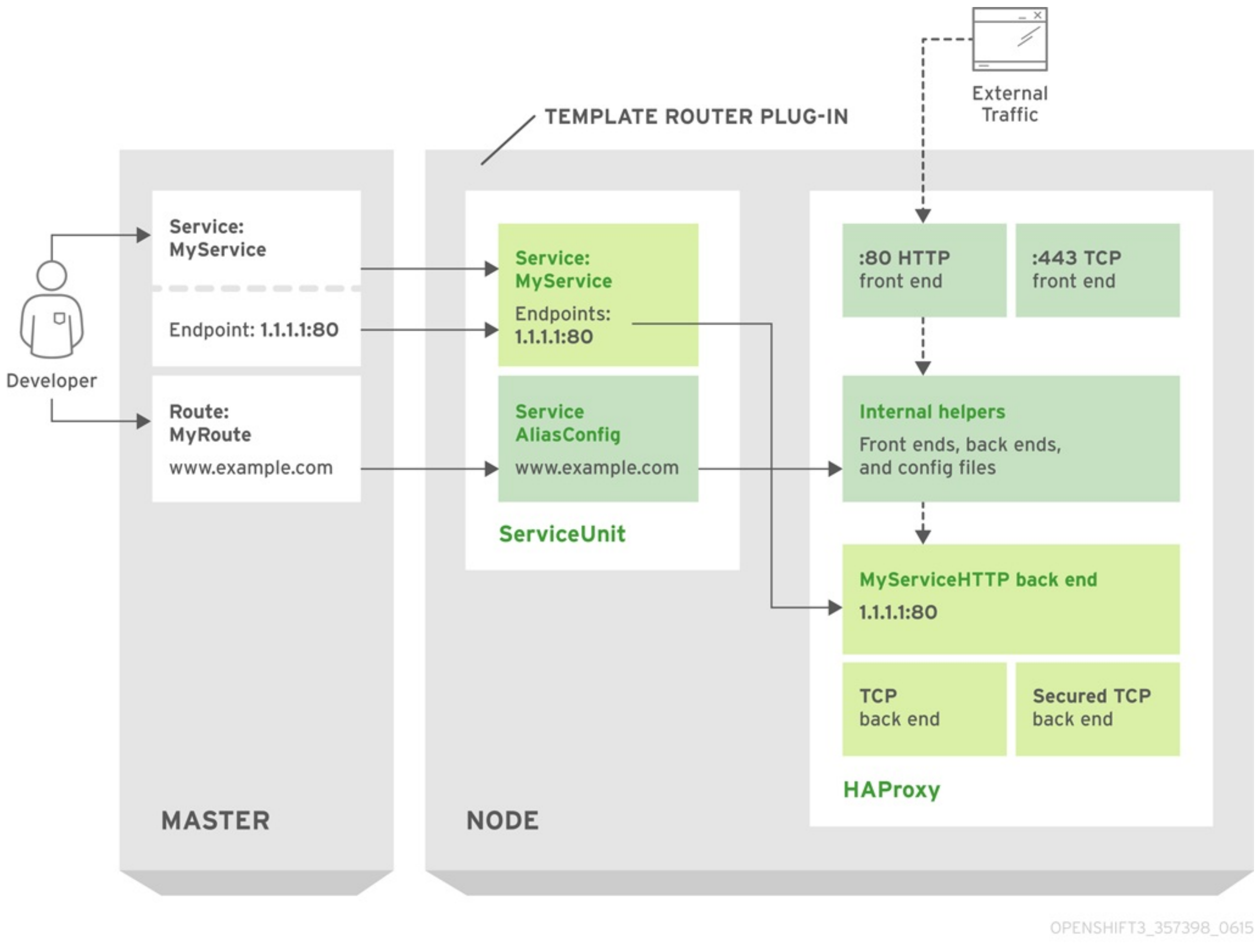
The following router plug-ins are provided and supported in OpenShift. Instructions on deploying these routers are available in Deploying a Router.

#### 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**



### Sticky Sessions

Implementing sticky sessions is up to the underlying router configuration. The default HAProxy template implements sticky sessions using the **`balance source`** directive which balances based on the source IP. 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.

#### 3.7.3.2. F5 Router

**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 feature parity with the HAProxy template router, which means it has feature parity, and then additional features, compared to to the **F5 BIG-IP®** support in OpenShift Enterprise 2. When comparing with the OpenShift **routing-daemon** used in earlier versions, the F5 router additionally supports:

- path-based routing (using policy rules),
- re-encryption (implemented using client and server SSL profiles), and
- 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.

### 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 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 **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.

### F5 Integration Details

The operation of the F5 router is similar to that of the OpenShift **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, 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, 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.

### 3.7.4. Route Host Names

In order for services to be exposed externally, an OpenShift 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 two routes claim the same host, the oldest route wins. If additional routes with different path fields are defined in the same namespace, those paths will be added. If multiple routes with the same path are used, the oldest takes priority.

**Example 3.6. 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 3.7. 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 automatically generates one for you. The generated host name is of the form:

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

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

**Example 3.8. 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.

## 3.7.5. 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 3.9. 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.

### 3.7.6. 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 hostname, 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 3.1. Route Availability**

| Route | When Compared to | Accessible |
|---|---|---|
| *www.example.com/test* | *www.example.com/test* | Yes |
| | *www.example.com* | No |
| *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 3.10. 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.

### 3.7.7. Secured Routes

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

**Note**

TLS termination in OpenShift 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 hostname, 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 3.11. 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.

**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 3.12. 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. <2>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).

**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 3.13. 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.

## 3.8. 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. 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.

**Example 3.14. A Simple Template Object Definition (YAML)**

```
apiVersion: v1
kind: Template
metadata:
  name: redis-template 1
  annotations:
    description: "Description" 2
    iconClass: "icon-redis" 3
    tags: "database,nosql" 4
objects:    5
- apiVersion: v1
  kind: Pod
  metadata:
    name: redis-master
  spec:
    containers:
    - env:
      - name: REDIS_PASSWORD
        value: ${REDIS_PASSWORD} 6
      image: dockerfile/redis
      name: master
      ports:
      - containerPort: 6379
        protocol: TCP
```

```
parameters:     7
- description: Password used for Redis authentication
  from: '[A-Z0-9]{8}'     8
  generate: expression
  name: REDIS_PASSWORD
labels:         9
  redis: master
```

1 The name of the template

2 Optional description for the template

3 The icon that will be shown in the UI for this template; the name of a CSS class defined in the web console source (search content for "openshift-logos-icon").

4 A list of arbitrary tags that this template will have in the UI

5 A list of objects the template will create (in this case, a single pod)

6 Parameter value that will be substituted during processing

7 A list of parameters for the template

8 An expression used to generate a random password if not specified

9 A list of labels to apply to all objects on create

▪

A template describes a set of related object definitions to be created together, as well as a set of parameters for those objects. For example, an application might consist of a frontend web application backed by a database; each consists of a service object and deployment configuration object, and they share a set of credentials (parameters) for the frontend to authenticate to the backend. The template can be processed, either specifying parameters or allowing them to be automatically generated (for example, a unique DB password), in order to instantiate the list of objects in the template as a cohesive application.

Templates can be processed from a definition in a file or from an existing OpenShift API object. Cluster administrators can define standard templates in the API that are available for all users to process, while users can define their own templates within their own projects.

Administrators and developers can interact with templates using the CLI and web console.

### 3.8.2. Parameters

Templates allow you to define parameters which take on a value. That value is then substituted wherever the parameter is referenced. References can be defined in any text field in the objects list field.

Each parameter describes a variable and the variable value which can be referenced in any text field in the **`objects`** list field. During processing, the value can be set explicitly or it can be generated by OpenShift.

An explicit value can be set as the parameter default using the **`value`** field:

```
parameters:
  - name: USERNAME
    description: "The user name for Joe"
    value: joe
```

The **`generate`** field can be set to 'expression' to specify generated values. The **`from`** field should specify the pattern for generating the value using a pseudo regular expression syntax:

```
parameters:
  - name: PASSWORD
    description: "The random user password"
    generate: expression
    from: "[a-zA-Z0-9]{12}"
```

In the example above, processing will generate a random password 12 characters long consisting of all upper and lowercase alphabet letters and numbers.

The syntax available is not a full regular expression syntax. However, you can use **`\w`**, **`\d`**, and **`\a`** modifiers:

- **`[\w]{10}`** produces 10 alphabet characters, numbers, and underscores. This follows the PCRE standard and is equal to **`[a-zA-Z0-9_]{10}`**.
- **`[\d]{10}`** produces 10 numbers. This is equal to **`[0-9]{10}`**.
- **`[\a]{10}`** produces 10 alphabetical characters. This is equal to **`[a-zA-Z]{10}`**.

# CHAPTER 4. ADDITIONAL CONCEPTS

## 4.1. NETWORKING

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.

### 4.1.1. OpenShift 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 has a built-in DNS so that the services can be reached by the service DNS as well as the service IP/port. OpenShift 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 Docker 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 Docker behavior. 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 |

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

This prevents having to restart frontend pods in order to pick up new services, which 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.

### 4.1.2. OpenShift SDN

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

OpenShift SDN is automatically installed and configured as part of the Ansible-based installation procedure. Further administration should not be required; however, further details on the design and operation of OpenShift SDN are provided for those who are curious or need to troubleshoot problems.

## 4.2. OPENSHIFT SDN

### 4.2.1. Overview

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

OpenShift SDN includes the **ovssubnet** SDN plug-in for configuring the network, which provides a "flat" pod network where every pod can communicate with every other pod and service.

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

### 4.2.2. Design on Masters

On an OpenShift 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 **10.1.0.0/16** class B network, and nodes are allocated **/24** subnets (i.e., **10.1.0.0/24**, **10.1.1.0/24**, **10.1.2.0/24**, and so on). This means that the cluster network has 256 subnets available to assign to nodes, and a given node is allocated 254 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 containers via the cluster network, unless it is also running as a node.

### 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 six network devices:

- **br0**, the OVS bridge device that containers will be attached to. OpenShift SDN also configures a set of non-subnet-specific flow rules on this bridge. The **ovssubnet** plug-in waits to do so until the SDN master announces the creation of the new node subnet.
- **lbr0**, a Linux bridge device, which is configured as Docker’s bridge and given the cluster subnet gateway address (eg, 10.1.x.1/24).
- **tun0**, an OVS internal port (port 2 on **br0**). This *also* 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.
- **vlinuxbr** and **vovsbr**, two Linux peer virtual Ethernet interfaces. **vlinuxbr** is added to **lbr0** and **vovsbr** is added to **br0** (port 9), to provide connectivity for containers created directly with Docker outside of OpenShift.
- **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. moves the host side of the pod’s veth interface pair from the **lbr0** bridge (where Docker placed it when starting the container) to the OVS bridge **br0**.
2. adds OpenFlow rules to the OVS database to route traffic addressed to the new pod to the correct OVS port.

The pod is allocated an IP address in the cluster subnet by Docker itself because Docker is told to use the **lbr0** bridge, which OpenShift SDN has assigned the cluster gateway address (eg. 10.1.x.1/24). Note that the **tun0** is also assigned the cluster gateway IP address because it is the default gateway for all traffic destined for external networks, but these two interfaces do not conflict because the **lbr0** interface is only used for IPAM and no OpenShift SDN pods are connected to it.

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.

#### 4.2.3.1. Packet Flow

Suppose we 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 Docker'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.

#### 4.2.3.2. External Access to the Cluster Network

If a host that is external to OpenShift requires access to the cluster network, you have two options:

1. Configure the host as an OpenShift node but mark it unschedulable so that the master does not schedule containers on it.

2. Create a tunnel between your host and a host that is on the cluster network.

Both options are presented as part of a practical use-case in the documentation for configuring routing from an edge load-balancer to containers within OpenShift SDN.

## 4.3. AUTHENTICATION

### 4.3.1. Overview

The authentication layer identifies the user associated with requests to the OpenShift 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.

### 4.3.2. Users and Groups

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

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 any currently-authenticated users. |
| **system:unauthenticated** | Automatically associated with any currently-unauthenticated users. |

### 4.3.3. API Authentication

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

**OAuth Access Tokens**

- Obtained from the OpenShift 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.

### 4.3.4. OAuth

The OpenShift 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.

**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 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
  ]
}')
```

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**.

### 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 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 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 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.

### 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 |

| Status | Content | Client response |
|---|---|---|
| 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 |

## 4.4. AUTHORIZATION

### 4.4.1. 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 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. |

|  |  |
|---|---|
| **Bindings** | Associations between users and/or groups with a role. |

Rules, roles, and bindings can be visualized 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]
```

### 4.4.2. Evaluating Authorization

Several factors are combined to make the decision when OpenShift 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`**, or **`watch`**. |
| Resource Name | The API endpoint being accessed. |

| | |
|---|---|
| **Bindings** | The full list of bindings. |

OpenShift 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.

### 4.4.3. 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.

### 4.4.4. Roles

Roles are collections of policy rules, which are sets of permitted verbs that can be performed on a set of resources. OpenShift 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.

These roles, including a matrix of the verbs and resources each are associated with, can be visualized in the cluster policy by using the CLI to view the cluster roles. Additional **system:** roles are listed as well, which are used for various OpenShift 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.

#### 4.4.4.1. Updating Cluster Roles

After any OpenShift cluster upgrade, the recommended default roles may have been updated. See the Administrator Guide for instructions on updating the policy definitions to the new recommendations using:

```
$ oadm policy reconcile-cluster-roles
```

### 4.4.5. Security Context Constraints

In addition to authorization policies that control what a user can do, OpenShift 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 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.

Two SCCs are added to the cluster by default, *privileged* and *restricted*, which are viewable by cluster administrators using the CLI:

```
$ oc get scc
NAME         PRIV      CAPS      HOSTDIR   SELINUX      RUNASUSER
privileged   true      []        true      RunAsAny     RunAsAny
restricted   false     []        false     MustRunAs    MustRunAsRange
```

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
allowPrivilegedContainer: true
apiVersion: v1
```

```
groups: (1)
- system:cluster-admins
- system:nodes
kind: SecurityContextConstraints
metadata:
  creationTimestamp: null
  name: privileged
runAsUser:
  type: RunAsAny (2)
seLinuxContext:
  type: RunAsAny (3)
users: (4)
- system:serviceaccount:openshift-infra:build-controller
```

1 The groups that have access to this SCC

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

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

4 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.

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.

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:<br>- A mechanism to generate the value, and<br>- A mechanism to ensure that a specified value falls into the set of allowable values. |

#### 4.4.5.1. 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 any security context setting that was 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.

## 4.5. PERSISTENT STORAGE

### 4.5.1. Overview

Managing storage is a distinct problem from managing compute resources. OpenShift leverages the Kubernetes **`PersistentVolume`** subsystem, which provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed. This subsystem uses the **`PersistentVolume`** and **`PersistentVolumeClaim`** API objects.

A **`PersistentVolume`** (PV) object 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.

A **`PersistentVolumeClaim`** (PVC) object represents a request for storage by a user. 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).

### 4.5.2. 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.

#### 4.5.2.1. 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.

#### 4.5.2.2. 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.

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.

#### 4.5.2.3. 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.

#### 4.5.2.4. 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.

#### 4.5.2.5. 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.

### 4.5.3. 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
```

#### 4.5.3.1. Types of Persistent Volumes

OpenShift Enterprise currently supports the following **`PersistentVolume`** plug-ins:

- NFS
- HostPath (single node testing only)

More plug-ins are available but are currently in Technology Preview:

- GCE Persistent Disks
- AWS Elastic Block Stores (EBS)
- GlusterFS
- iSCSI

- RBD (Ceph Block Device)

#### 4.5.3.2. 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.

#### 4.5.3.3. 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.

The access modes are:

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

**Important**

A volume can only be mounted using one access mode at a time, even if it supports many. For example, a GCE Persistent Disk can be mounted as **ReadWriteOnce** by a single node or **ReadOnlyMany** by many nodes, but not at the same time.

#### 4.5.3.4. Recycling Policy

The current recycling policies are:

| Recycling Policy | Description |
|---|---|
| Retain | Manual reclamation |

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

Currently, NFS and HostPath support recycling.

#### 4.5.3.5. 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.

### 4.5.4. 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
```

#### 4.5.4.1. Access Modes

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

#### 4.5.4.2. 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.

#### 4.5.4.3. 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
```

## 4.6. REMOTE COMMANDS

### 4.6.1. Overview

OpenShift 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.

### 4.6.2. 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 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.

## 4.7. PORT FORWARDING

### 4.7.1. Overview

OpenShift 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.

### 4.7.2. 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 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.

## 4.8. THROTTLING

### 4.8.1. Overview

OpenShift clusters will orchestrate many potentially large applications that could be co-located on a set of shared nodes. Throttling refers to the act of controlling pod start order and resource consumption to provide:

1. Optimal start-up time when the system has to start large numbers of pods at once
2. Resource control so that a single container cannot monopolize the resources of an entire node

## 4.9. SOURCE CONTROL MANAGEMENT

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

- Triggering a build upon repository changes.

## 4.10. OTHER API OBJECTS

### 4.10.1. 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.

### 4.10.2. 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 the Developer’s Guide and Kubernetes documentation for more information on **`ResourceQuota`**.

### 4.10.3. 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’s Guide and Kubernetes documentation for more information.

### 4.10.4. 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.

### 4.10.5. 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.

### 4.10.6. 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.

### 4.10.7. OAuth Objects

#### 4.10.7.1. 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**.

#### 4.10.7.2. 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": []
}
```

#### 4.10.7.3. 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.

#### 4.10.7.4. 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.

### 4.10.8. User Objects

#### 4.10.8.1. Identity

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

OpenShift 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.

#### 4.10.8.2. 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.

### 4.10.8.3. 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

### 4.10.8.4. 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

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

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. THU MAY 19 2016

| Affected Topic | Description of Change |
|---|---|
| Core Concepts → Builds and Image Streams | Updated the example in the Image Stream Mappings section to use `https` for GitHub access. |

## 5.2. TUE MAY 03 2016

| Affected Topic | Description of Change |
|---|---|
| Infrastructure Components → Web Console | Added a link to the OpenShift and Atomic Enterprise Platform Tested Integrations article in the Browser Requirements section. |

## 5.3. WED APR 27 2016

| Affected Topic | Description of Change |
|---|---|
| Core Concepts → Routes | Corrected footnote placement in the Secured Routes section. |
| Core Concepts → Builds and Image Streams | Added information about maximum name lengths for objects. |
| Core Concepts → Projects and Users | Added information about maximum name lengths for objects. |
| Core Concepts → Pods and Services | Added information about maximum name lengths for objects. |

## 5.4. TUE APR 19 2016

| Affected Topic | Description of Change |
|---|---|
| Infrastructure Components → Web Console | Added a Browser Requirements section, which outlines browser versions and operating systems that can be used to access the web console. |

## 5.5. MON APR 04 2016

| Affected Topic | Description of Change |
|---|---|
| Core Concepts → Routes | Fixed typo of the **destinationCACertificate** parameter name. |

## 5.6. THU FEB 25 2016

| Affected Topic | Description of Change |
|---|---|
| Infrastructure Components → Kubernetes Infrastructure | Added a note indicating that moving from a single master cluster to multiple masters after installation is not supported. |

## 5.7. MON FEB 15 2016

| Affected Topic | Description of Change |
|---|---|
| Core Concepts → Routes | Updated to clarify that **`destinationCaCertificate`** is required, but only for re-encryption. |

## 5.8. MON FEB 01 2016

| Affected Topic | Description of Change |
|---|---|
| Core Concepts → Builds and Image Streams | Added more information on how builds work behind the scenes. |

| Affected Topic | Description of Change |
|---|---|
| Additional Concepts → Persistent Storage | Added an Important box about providing high-availability. |

## 5.9. TUE JUN 23 2015

OpenShift Enterprise 3.0 release.