© William Caban 2019
W. Caban, *Architecting and Operating OpenShift Clusters*
https://doi.org/10.1007/978-1-4842-4985-7_1

1. The OpenShift Architecture

William Caban[1]

(1) Columbia, MD, USA

To properly architect an OpenShift cluster, we need to understand the different components of the platform, their roles, and how they interact with each other. This base knowledge is important to be able to fine-tune OpenShift cluster design to your organization's need beyond what is covered in this book.

Before going into each main component of OpenShift, it is important to understand how it relates to *Linux Containers* and *Kubernetes*.

Linux Containers

Nowadays, when we see the term *Linux Containers,* it is easy to think it only refers to *Docker,* but that is not the case. The term *Linux Containers* denotes a group of technologies used to package and isolate applications, their dependencies, and their runtimes, in a portable way so it can be moved between environments while retaining full functionality.

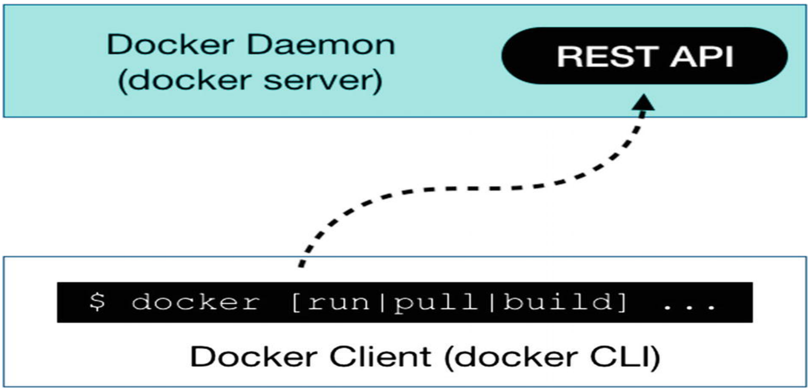
A source of confusion is because the term *Docker* refers to various elements of a technology that popularized the *Linux Containers*.

First, there is *Docker Inc.*, the name of the company that popularized the Linux Containers technology with the *Docker* platform. The *Docker* platform was originally built as a series of enhancements on top of the LXC technology to bring better isolation and portability to the containers.

Second, there is *Docker Daemon* which is the daemon or service that serves the Docker API, handles the API requests, and manages images, containers, networks, and volumes.

Finally, there are *Images* and *Containers* respectively referred to as the *Docker Images* and *Docker Containers*. The *Image* is the read-only template containing the application, the application dependencies, and the required runtime environment. All this packaged in a standard format used by the *Docker Daemon* . The *Container* refers to a runnable instance of an *Image*.

As it can be seen in Figure 1-1, *Docker* is a client-server application to build and run containers following a standardized container format. The *docker* client is the tool used to interact with the docker server over the API exposed by the *Docker Daemon.*



```
$ docker [run|pull|build] ...
```

Figure 1-1 The Docker client-server architecture

NOTE The terms *Docker Daemon* and *Docker Engine* are used interchangeably to refer to the docker server.

LINUX CONTAINER: UNDER THE HOOD

Beyond the conceptual definitions of containers as an artifact containing an application and all its dependencies, or as an artifact that is built once and deployed "anywhere," what is a Linux Container?

To understand containers and how they work, we must explore some important building blocks at the Linux Kernel: *namespaces* and *cgroups*.

Linux *namespaces* provide process isolation. There are seven[1] kinds of Kernel namespaces:

- **Mount**: The mount namespace isolates the set of filesystem mount points. This enables the creation of different views of the filesystem hierarchy or making certain mount points read-only for processes in different mount namespaces.

- **UTC**: This namespace enables for each container to have its own *hostname* and NIS domain name.

- **IPC**: This isolates interprocess communication (IPC) resources between namespaces. This enables more than one container to create shared memory segments and semaphores with the same name but is not able to interact with other containers' memory segments or shared memory.

- **PID**: Each process receives PID namespace provided. The container only sees the processes within the container and not any processes on the host or other containers.

- **Network**: This allows the container to only communicate with internal or external networks. This provides a loopback interface as the initial network interface. Additional physical or virtual network interfaces can be added to the namespace. Each namespace maintains a set of IP addresses and its own routing table.

- **User**: This isolates the *user* IDs between namespaces providing privilege isolation and *user* ID segregation.

- **Control Group (cgroup)** (the namespace): This virtualizes the view of *cgroups* for a set of processes enabling better confinement of containerized processes.

The namespaces are Kernel-level capabilities. As such, each namespace has visibility about all the host capabilities and system resources. Namespaces isolate system resources by providing an abstraction layer for the processes inside the namespaces. It does this by creating a view where it appears as the processes have the global resources.

A way to think about *namespaces* is going back to our science fiction world of *parallel universes*. Each *namespace* is a parallel reality or a parallel universe inside the universe represented by the host. These parallel universes do not know of the existence of any other universe and cannot interfere with them.

Now, if each namespace has a full view of the host system resources, by default, it could assume it can consume all the resources it detects, for example, all the memory and all the CPU resources. To limit the access to the system resources is the functionality of the next building block: *Control Groups* or *cgroups*.

Control Groups (cgroups), the Kernel feature, are used for limiting, accounting, and controlling resources (i.e., memory, CPU, I/O, check pointing) of a collection of processes. A container is a collection of processes under a PID namespace. To control and limit resources for a container, we use *cgroups*.

Bringing all these concepts together, we can visualize containers as illustrated in Figure 1-2.

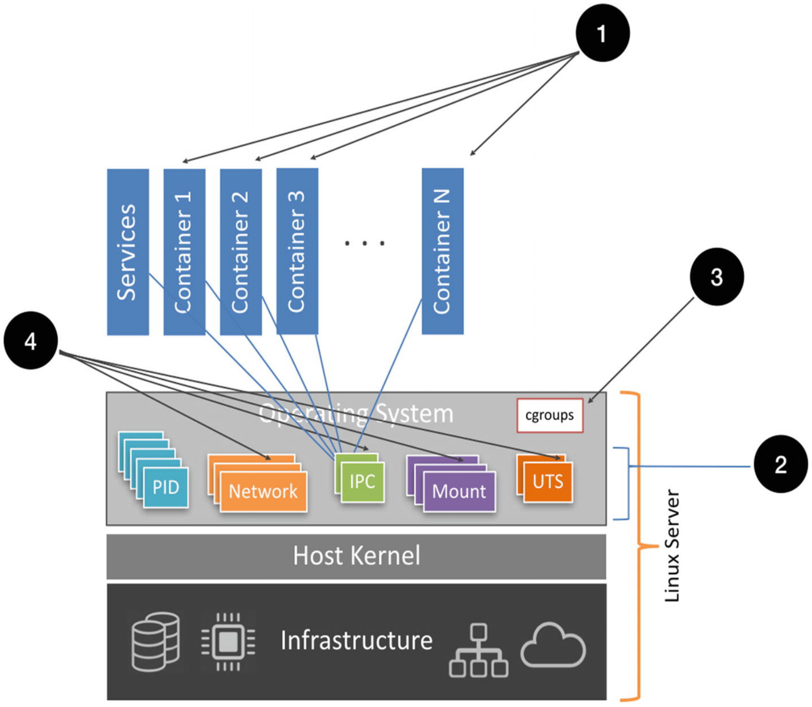


Figure 1-2 Linux namespaces and Containers

To explain the details of Figure 1-2, follow the numbers from the illustration with the numbers of this list:

1. Each *Container* has a unique PID namespace running its group of process. Inside the Container, the first process is seen as PID 1. From the host perspective, the *Container PID* is a regular process ID.

2. The *Namespaces* exist at the Kernel level. Namespaces provide the isolation for the system resource but are part of the same Kernel.

3. *Control Groups* or *cgroups*, the feature, are used to limit the access to system resources by a group of processes.

4. In addition to the PID namespace, *Containers* will have other dedicated namespaces which provide their view of system resources or they can use the *default namespace* which is shared with the host.

CONTAINER SPECIFICATIONS

As can be seen from the previous section, from the technical perspective, in its core, Linux containers are a group of Linux processes existing in *namespaces* using *cgroups* to control and limit the access to system resources.

The core building blocks for Linux Containers are simple but powerful. Following the popularity of *Docker* containers, the industry recognized the need for a set of specifications (Figure 1-3) supported by open communities to maintain compatibility while enabling innovation and creation of solutions on top of the capabilities provided by Linux Containers.

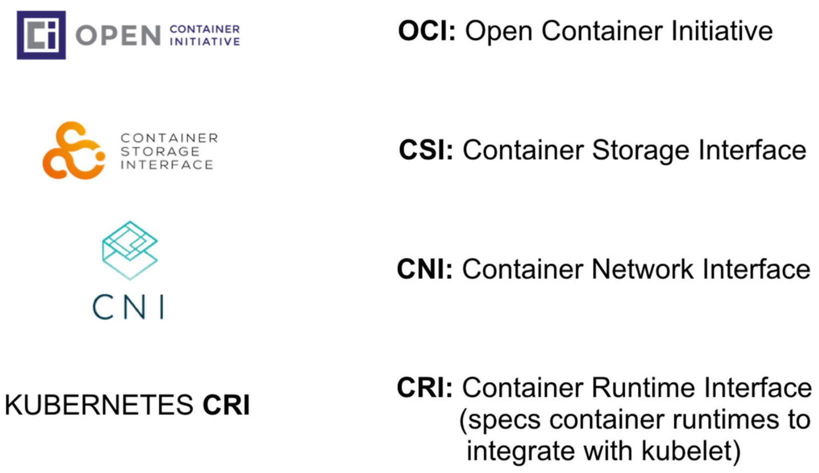

OPEN CONTAINER INITIATIVE
OCI: Open Container Initiative
CONTAINER STORAGE INTERFACE
CSI: Container Storage Interface
CNI
CNI: Container Network Interface
KUBERNETES CRI
CRI: Container Runtime Interface (specs container runtimes to integrate with kubelet)

Figure 1-3 The Container specifications

Today, the widely recognized container specifications are

1. **Open Container Initiative (OCI)** : The OCI specification defines a *standard* container format. This is what is usually referred as the *Docker format* (Figure 1-4).

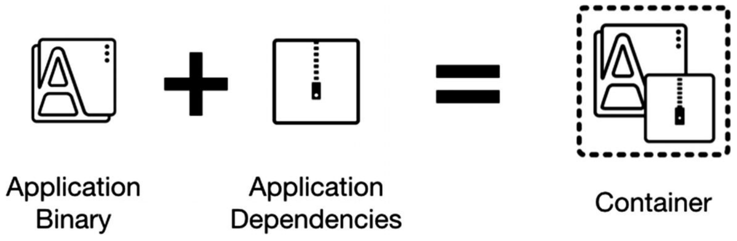


Figure 1-4 The OCI-compliant container image

NOTE Demystifying containers—An OCI-compliant image, or an image following the *Docker format*, can be seen as a TAR file of a filesystem layout containing the application binaries, its dependencies, and some XML formatted files with metadata describing the container namespaces. A container with multiple layers is a TAR file of TAR files, each representing a particular layer of the container.

2. **Container Storage Interface (CSI)** : The CSI specification describes a standardized interface to present storage resources to a container. Prior to this specification, each storage vendor had to create, certify, and maintain their own storage plugin for every container solution. With CSI, vendors maintain a single plugin which can be used by any container solution supporting the specification.

3. **Container Network Interface (CNI)** : The CNI specification standardizes an interface to provide networking services to containers. This helped in reducing the proliferation of networking plugins which were incompatible among themselves.

CONTAINER RUNTIME AND KUBERNETES

The creation of the OCI specification also provided the freedom to replace the container runtime beyond the *Docker Daemon*. A *container runtime* only needs to understand the OCI format to be able to run the container.

Traditionally, by default, container runtime like the *Docker Daemon* handles containers in a single host. Over time, some of these tools evolved into fat daemons or services trying to include container orchestration and to solve and handle too many things (resource consumptions, scheduling, control, etc.).

NOTE For the remaining of this book, we use the term Linux Container, Containers, and Container Images to refer to a Linux Container following the OCI specification.

With Kubernetes, Google provided a way to orchestrate, manage, and operate containers at scale across thousands of nodes. Kubernetes abstracted the management of individual containers with the notion of managing *Pods* and *Services*. Kubernetes, as the container orchestration platform, requires minimal actions to be handled by the container runtimes: create Pod, start Pod, stop Pod, and remove Pod.

With this new understanding, the Kubernetes community explored ways to replace traditional fat daemons with purpose built container runtimes. The community defined the **Container Runtime Interface (CRI) . CRI**[2] provides a specification for integrating container runtimes with the *kubelet* service at each Kubernetes worker node. Since then, there has been a proliferation of CRI-compliant container runtimes for Kubernetes optimizing for speed, isolation, and breaking dependencies to a runtime daemon. Among these new options, we can find containerd, Kata Containers, and CRI-O.

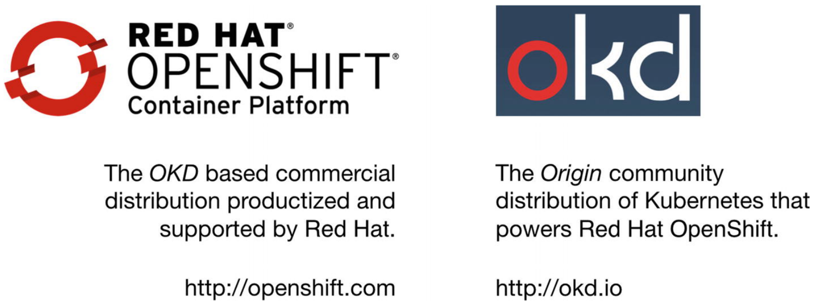
NOTE OpenShift 3.x supports the *Docker Daemon* as the default container runtime. Starting with OpenShift 3.10, it also supports CRI-O as the container runtime. With OpenShift 4.0, CRI-O will be the default container runtime.

Introduction to OpenShift Architecture Components

OpenShift is built on top of Kubernetes. While Kubernetes provides the container orchestration capabilities, *Pod* resiliency, *Services* definitions, and

Deployment constructs to describe the desire state of a microservice-based application, there are many other components required to make it work. For example, Kubernetes does not provide a default Software-Defined Networking (*SDN)* or a default method to steer traffic into the applications running on Kubernetes clusters. It is up to the cluster admin to bring additional tools and projects to operate and manage the Kubernetes cluster and any application running on it. For the developers it also means they need to learn a new CLI or YAML specification to be able to deploy and test their applications. For the security teams, it means figuring out ways to map the organization's policies into new constructs and identifying additional projects to enforce additional ones not provided by the default capabilities of Kubernetes.

These additional capabilities are part of what is provided out of the box with OpenShift Container Platform or OKD (the upstream community project) (see Figure 1-5). In fact, at the time of this writing, OpenShift is a Kubernetes superset combining over 200 open source projects into a fully integrated solution with strong focus on a developer experience, operational capabilities, monitoring, and management with strong and secure defaults. All these while being pluggable so platform admins can replace out of the box components and services with their own. For example, using third-party SDN to provide the networking capabilities or third-party storage solutions to provide persistent storage for the applications running in the environment.



The *OKD* based commercial distribution productized and supported by Red Hat.

http://openshift.com

The *Origin* community distribution of Kubernetes that powers Red Hat OpenShift.

http://okd.io

Figure 1-5 OpenShift Container Platform (OCP) vs. OKD (formerly OpenShift Origin)

NOTE In this book the term OpenShift is used to denote both the OpenShift Container Platform (OCP) , which is the Red Hat–supported product, and OKD, the upstream community project. Unless otherwise specified, everything in this book applies to OCP and OKD.

Kubernetes Constructs

Having Kubernetes as its foundation, OpenShift inherits all the base constructs for the *Containers'* orchestration from Kubernetes and, in addition, extends them. A great deal of these extensions come from adding the capabilities or functionalities that are not part of the base of Kubernetes but that are required to successfully operate the platform. Other extensions come from enforcing prescriptive best practices designed to comply with the stability and regulations required on enterprise environments (i.e., RBAC, CI/CD Pipelines, etc.).

Some of the important Kubernetes constructs inherited by OpenShift (not an exhaustive list) are

- ***Pods*** : A *Pod* is a group of one or more tightly coupled *Containers* sharing a set of Linux namespaces and cgroups (Figure 1-6). Among those, the *Containers* inside the Pod share the same *Mount* and *Network* namespace (i.e., same IP address and TCP/UDP ports) (see per-Pod IP addresses in Figure 1-6). Within a *Pod* each *Container* may have further sub-isolations (i.e., different UTC namespaces). *Pods* communicate with each other using *localhost*.

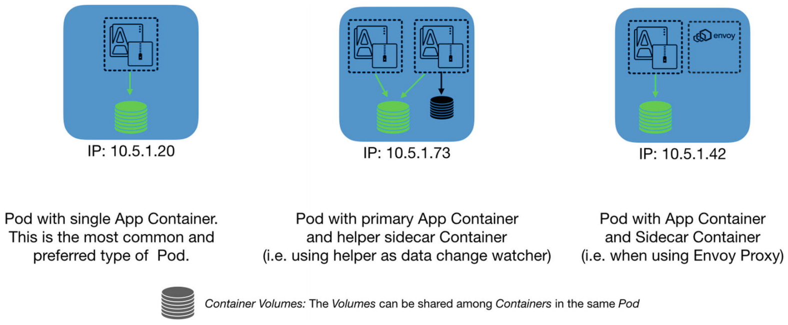

envoy
IP: 10.5.1.20
IP: 10.5.1.73
IP: 10.5.1.42
Pod with single App Container. This is the most common and preferred type of Pod.
Pod with primary App Container and helper sidecar Container (i.e. using helper as data change watcher)
Pod with App Container and Sidecar Container (i.e. when using Envoy Proxy)
Container Volumes: The Volumes can be shared among Containers in the same Pod

Figure 1-6 Example of Pod configurations

- ***Services***: A *Service* is a Kubernetes object that maps one or more incoming ports to *targetPorts* at a selected set of target of *Pods.* These represent a microservice or an application running on the cluster. The *Services* are discoverable by *Pods* running on the cluster. Generally, *Pods* interact with other applications or microservice on the cluster through the *Service* object (Figure 1-7).

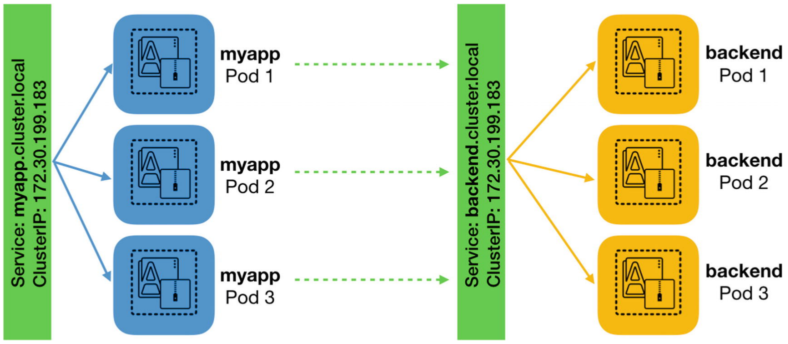

Service: myapp.cluster.local
ClusterIP: 172.30.199.183
myapp Pod 1
myapp Pod 2
myapp Pod 3
Service: backend.cluster.local
ClusterIP: 172.30.199.183
backend Pod 1
backend Pod 2
backend Pod 3

Figure 1-7 The Kubernetes Service object abstracts one or more Pods running an application or microservice

- ***ReplicationController (RC)*** : The *ReplicationController* (the object) ensures the requested number of *Pods* are running at any given time. If there are too many *Pods*, the *ReplicationController* terminates any number of *Pods* in excess of the specified amount. If there are too few, the *ReplicationController* starts additional *Pods* until the specified amount. In case of a *Pod* failure, or if *Pods* are deleted or terminated, the *ReplicationController* takes care of re-creating the failed, deleted, or terminated *Pods* to match the requested number of *Pods*.

- ***ReplicaSets***: The *ReplicaSets* are considered the next generation of *ReplicationControllers.* From the high-level perspective, *ReplicaSets* provide the same functionalities as the *ReplicationControllers* with the difference being these are intended to be managed by *Deployments*.

- ***Deployment*** *(the object)*: The *Deployment* object is a declarative configuration describing the desired state, quantity, and version of Pods to deploy. The *Deployment controller* defines a *ReplicaSet* that creates new *Pods* or executes a rolling upgrade with the new version of the *Pods*. The *Deployment Controller* changes and maintains the state of the *Pod* and *ReplicaSet* to match the desire state (Figure 1-8).

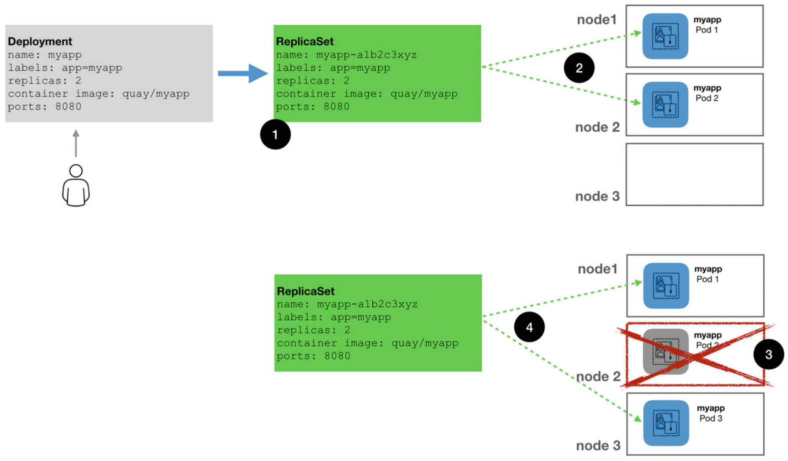


Figure 1-8 Deployment and ReplicaSet

- The steps illustrated by the pseudocode in Figure 1-8 are as follows:

  1. The *Deployment* object creates a *ReplicaSet* with the information of the desired state.

  2. The *ReplicaSet* deploys the requested version and total number of *Pods.*

  3. In case of *Pod* failure (i.e., because of node failure), the total number of *Pods* will be less than the desired amount.

  4. The *ReplicaSet* will deploy additional Pods until the number of desired replicas specified by the *Deployment.*

- ***Volumes***: The *Volumes* provide persistent storage for the Containers inside a *Pod*. Data in a *Volume* is preserved across Container restarts. *Volumes* outlive *Containers* and remain in existence for the lifetime of a *Pod*.

- ***PersistentVolume (PV)***: The *PersistentVolume* represents the actual storage resource provisioned for the cluster. *PVs* are *Volume* plugins with a lifecycle independent of any *Pod* that uses the *PV*.

- ***PersistentVolumeClaim (PVC)*** : The *PersistentVolumeClaim* is the storage request for the PV storage resources. A *PVC* is bind to a *PV* matching the requested storage characteristics and access mode. Refer to Figure 1-9.

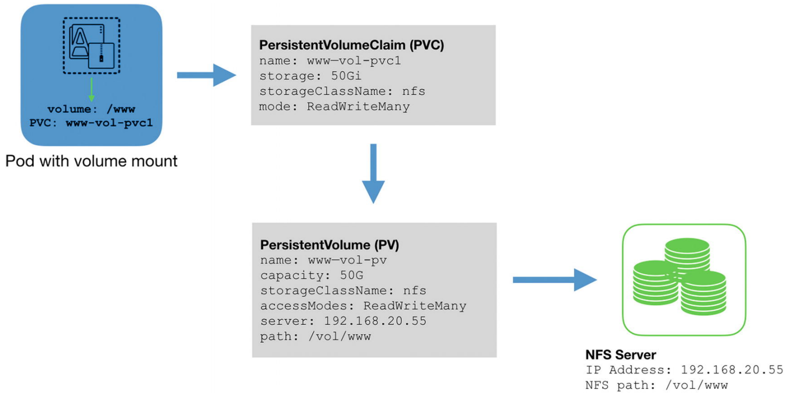


Figure 1-9 Relationship between Volume, PersistentVolumeClaim, and PersistentVolume

The Kubernetes architecture is comprised of the following core elements (not an exhaustive list):

- **Master Nodes**: The master nodes are the nodes hosting core elements of the control plane like (not an exhaustive list) the *kube-api-server*, *kube-scheduler, kube-controller-manager,* and in many instances the *etcd* database.
  - ***kube-api-server***: This component is what is commonly referred as the Kubernetes API. This is the frontend API to the control plane of the Kubernetes cluster.
  - ***kube-scheduler***: This component takes care of handling the scheduling of Pods into nodes, taking into account resource requirements, policy constraints, affinity or anti-affinity rules, and other filters.
  - ***kube-controller-manager***: This component runs multiple controller services at the master. Among these controllers, we can find (not an exhaustive list)
    - ***Node Controller***: This controller is responsible for detecting node failures and triggering the appropriate response.
    - ***Replication Controller*** *(the controller)*: This controller is responsible for ensuring the correct number of Pods are running as requested by a *replication controller* (the object) in the system.
    - ***Endpoints Controller***: This manages the Endpoint objects by associating the correct Services and Pods.

    - ***etcd***: This component is a key-value store database used extensively by Kubernetes to store configuration data of the cluster representing the state of the cluster (i.e., nodes, pods state, etc.) as well as for service discovery, among other things.

- **Worker Nodes**: The worker nodes (formerly known as *minions*) host elements like the *kubelet*, *kube-proxy,* and the *container runtime*.

    - ***kubelet***: Also known as the *node agent*, is the Kubernetes agent that runs on each node. The *kubelet* ensures containers are started and continue to run as specified by the *container manifest* (a YAML file describing a Pod) and updates the node accordingly.

    - ***kube-proxy***: A simple Kubernetes network proxy agent running on each node. The *kube-proxy* abstracts network services defined on the host, forwards traffic to the appropriate *Service,* and provides traffic load balancing. It does this by managing *iptables* rules of the host.

    - ***A Container Runtime***: Any CRI-compliant runtime capable of running OCI-compliant *Containers* (i.e., Docker Daemon, CRI-O, containerd, etc.).

- **Ancillary Services**: Services required for the proper operation of the Kubernetes cluster but that are not technically considered to be part of the Kubernetes components. These services may be running as part of the *Master Nodes*, *Worker Nodes*, or dedicated *Nodes*, or even be services external to the cluster. Among these services, we can find DNS (i.e., SkyDNS or KubeDNS), Web UI Dashboards, container resource monitoring services, and cluster-level monitoring and logging services.

Figure 1-10 illustrates how all these elements integrate and interact to form the Kubernetes architecture.

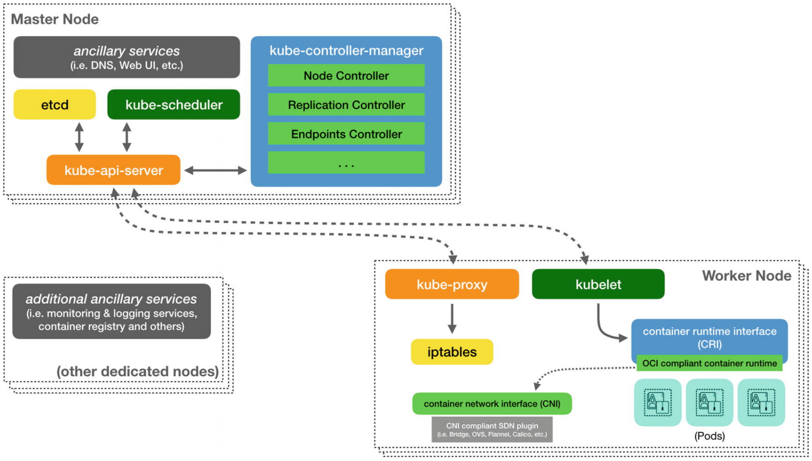


Figure 1-10 The elements of the Kubernetes architecture

OpenShift Constructs

The OpenShift architecture builds on top of Kubernetes and is comprised of three types of nodes:

- **Master Nodes**: These nodes are *Kubernetes Master Nodes* which may be providing additional functionalities like the web console with the self-service portal as well as the developers and operations-focused dashboards.

- **Infrastructure Nodes**: These are *Kubernetes Worker Nodes* dedicated to host functionalities like the *OpenShift Routes* and the OpenShift internal registry.

- **App Nodes or Nodes**: These are the *Kubernetes Worker Nodes* used to run the microservices and containerized applications deployed on OpenShift.

NOTE The *App Nodes* are also referred to just as *Nodes* and you will find them as such in some documentation. To avoid confusion, the book uses *App Nodes.*

As a superset of Kubernetes, within these nodes, beyond the Kubernetes elements, there can be multiple integrated components from other *Open Source* projects that work together to augment Kubernetes features and capabilities and form the OpenShift Container Platform. A special focus of this integration is toward the ease of use for developers and application owners.

A high-level view of the OpenShift node types is shown in Figure 1-11, and more details are going to be covered in subsequent sections.

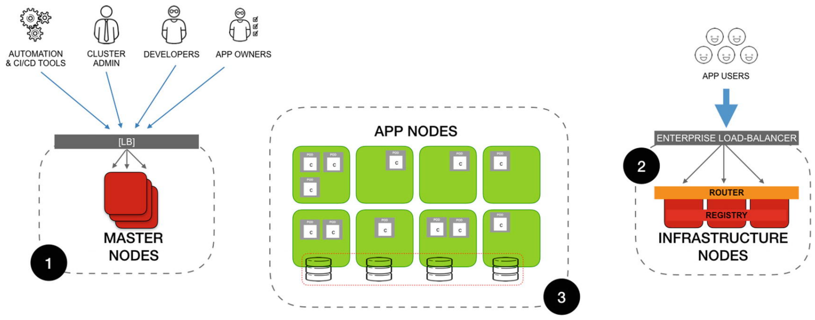


Figure 1-11 The OpenShift node types

Master Nodes

The *Master Nodes* are the main control elements of the OpenShift control plane. These are Kubernetes Master Nodes and they provide the services expected from any Kubernetes Master and additionally provide a series of functionalities built on top of Kubernetes which create OpenShift. See Figure 1-12 for reference.

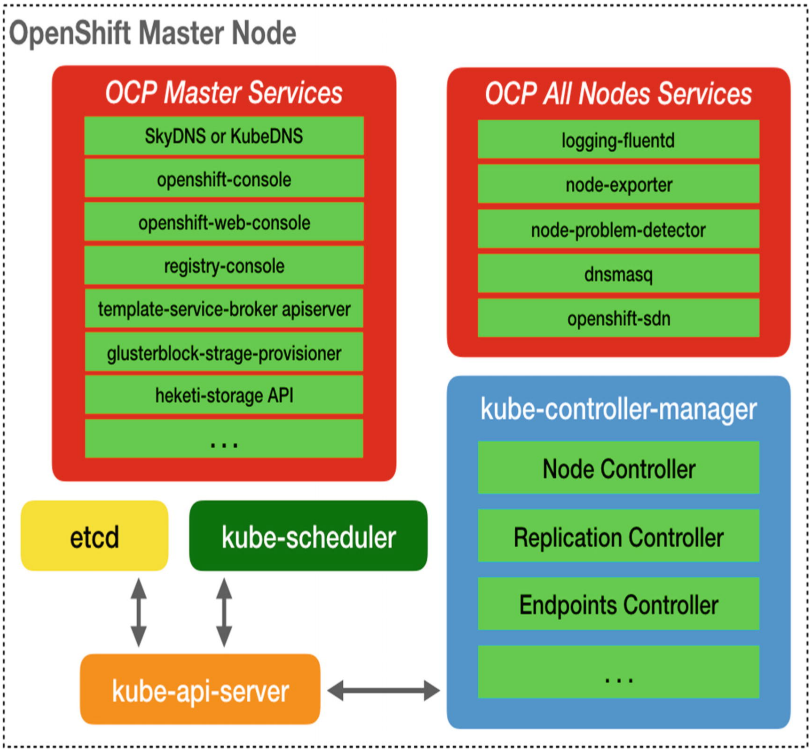


Figure 1-12 OpenShift Master Node details

From Figure 1-12 we can see the *Kubernetes Master Node* elements are present in the *OpenShift Master Nodes*. The actual list of these will be dependent on the services enabled for the cluster as many are optional services.

- **Kubernetes DNS**: The OpenShift 3.x releases are using SkyDNS as part of Kube-DNS. As of the writing of this book, this is transitioning to CoreDNS. By default, this DNS service listens on port 8053.

- **OpenShift Web Console**: This is the microservice providing the self-service portal or developer console.

- **OpenShift Console**: This is the microservice providing the operations console (former Tectonic console).

- **Registry Console**: This is the microservice providing a basic web UI to interact with the internal container registry.

- Additional **APIs** and **Consoles**: Many optional cluster services have their own API interfaces and web frontends. These APIs and frontends are provided as containers which, by default, will be hosted in the *Master Nodes.* Some examples are Template Service Brokers and the OpenShift Container Storage (glusterfs).

In addition, in Figure 1-12 we can see a partial list of services that may be present in every OpenShift Node. Their presence depends on the services enabled for the cluster. Let's go into the details of some of them (the actual service names may have slightly variations from the containers or Pods name):

- ***Fluentd***: The Fluentd service runs in every node. It aggregates logs from the host Node, including logs from *Pods* and *Projects*, and sends them to the *Elasticsearch (ES)* database running on the *Infrastructure Nodes*.

- ***node-exporter and kube-state-metrics***: These services are part of the OpenShift cluster monitoring solution based on Prometheus. The *node-exporter*[3] agent collects node hardware and OS metrics and makes them available for Prometheus. The *kube-state-metrics* agent converts metrics from Kubernetes objects (i.e., from the *kubelet*) into metrics consumable by Prometheus.

- ***node-problem-detector***: This is a service that runs in each node to detect multiple problems[4] on the node and reports them to the API Server.

- ***dnsmasq***: As part of the Kube-DNS service, this service is automatically configured on all nodes. Pods use the node hosting them as their default DNS. When receiving a name resolution request, dnsmasq will send the query to the *Kubernetes DNS* at the *Master Nodes,* and if not a resolution, it will try recursive DNS to the upstream DNS server originally configured on the node.

NOTE Every node is running *dnsmasq* listening on port 53. For this reason, nodes cannot run any other type of DNS application.

- ***openshift-sdn***: This consists of a series of privileged containers providing the Software-Defined Network (SDN) of the OpenShift cluster using Open vSwitch (OVS).

Infrastructure Nodes

These are dedicated *Kubernetes Worker Nodes* hosting important elements for the proper operation of the OpenShift Cluster. Among these, we have the *Container Registry* and the *OpenShift Router.* Figure 1-13 illustrates some additional services running on the *Infrastructure Nodes*.

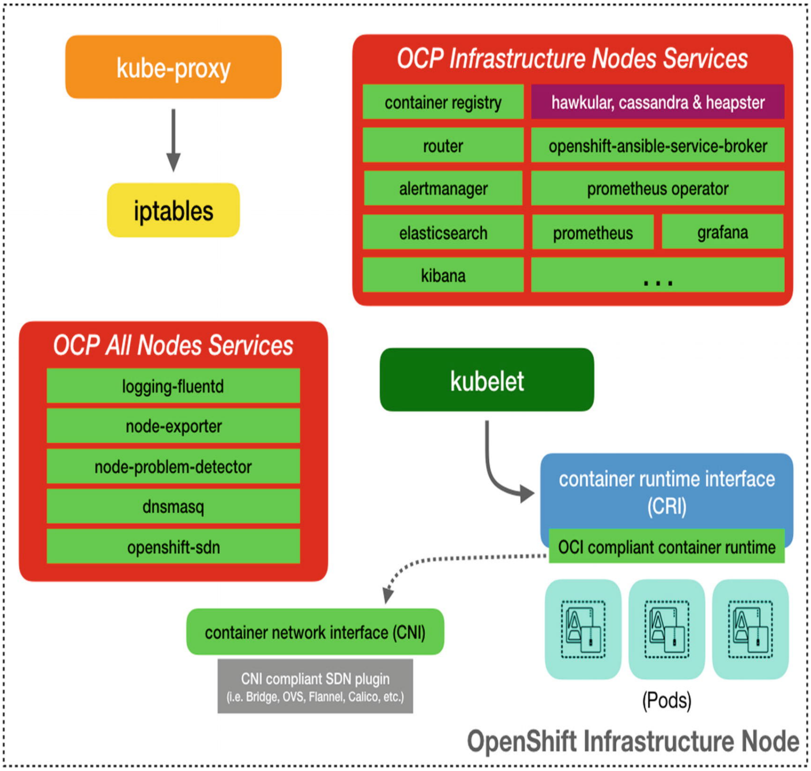

kube-proxy
iptables
OCP Infrastructure Nodes Services
container registry
hawkular, cassandra & heapster
router
openshift-ansible-service-broker
alertmanager
prometheus operator
elasticsearch
prometheus
grafana
kibana
...
OCP All Nodes Services
logging-fluentd
node-exporter
node-problem-detector
dnsmasq
openshift-sdn
kubelet
container runtime interface (CRI)
OCI compliant container runtime
container network interface (CNI)
CNI compliant SDN plugin
(i.e. Bridge, OVS, Flannel, Calico, etc.)
(Pods)
OpenShift Infrastructure Node

Figure 1-13 OpenShift Infrastructure Node details

From the diagram in Figure 1-13, we can deduce on *Infrastructure Nodes* there are some services which seem to overlap in functionalities. This is the case with services like *Hawkular, Cassandra, and Heapster* which are being deprecated in OpenShift 3.11 and being replaced by the Prometheus-based monitoring solution which is deployed and managed by the Prometheus Operator.

As with the *Master Nodes*, the exact list of services running on the *Infrastructure Nodes* is completely dependent on the services enabled for the cluster. Out of the services shown in the illustration, only few deserve mention at this point:

- ***OpenShift Container Registry (OCR)*** : The *OpenShift Container Registry* is a containerized *Docker Registry* service used internally by the cluster. Additional details are covered in the corresponding section.

- ***OpenShift Router***: The *OpenShift Router* is used to expose a *Kubernetes Service* to external clients by a *FQDN*. Additional details are covered in the corresponding section.

- ***Elasticsearch (ES) and Kibana***: *Elasticsearch* is used to collect all the logs sent by the *Fluentd* service running in every node. The *Kibana Web UI* is used to interact with the data and create visualization and dashboards of the aggregated data.

- ***Prometheus***, ***Grafana***, and the ***Prometheus Operator***: These are the components of the new *OpenShift Monitoring* and *Metrics* solution. These are used to collect information about the health of the cluster and all the services and components running on it. The *Grafana Web UI* is used to create dashboards visualizing the status of the elements being monitored.

App Nodes

The OpenShift App Nodes, or simply *OpenShift Nodes*, are *Kubernetes Worker Nodes* dedicated to running the workloads deployed to an OpenShift cluster. These include applications, microservices, or containerized applications.

As it can be seen in Figure 1-14, the OpenShift App Nodes are dedicated to running the applications deployed on the OpenShift cluster. Beyond the elements of the Kubernetes Nodes, it contains the common OpenShift Services to provide the network connectivity for the Pods, the DNS resolution, node monitoring, and log aggregation.

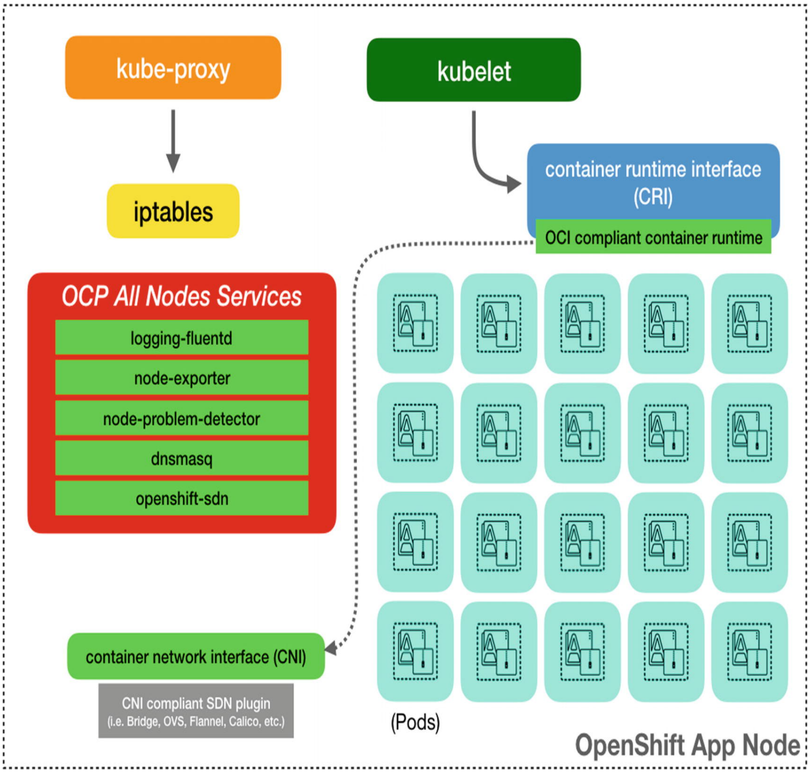


Figure 1-14 OpenShift App Node details

OpenShift Consoles

OpenShift provides developer-centric consoles and operations-centric consoles. The first console a user of the platforms receives is the *Service Catalog console* (see Figure 1-15) which contains the self-service catalog of pre-approved container images and templates (see #2 of Figure 1-15) available for the particular user. These catalogs can be cluster-wide catalogs or project-specific catalogs. From this initial console, the user can choose from a drop-down menu (see #1 of Figure 1-15) to switch to the operations *Cluster Console*.

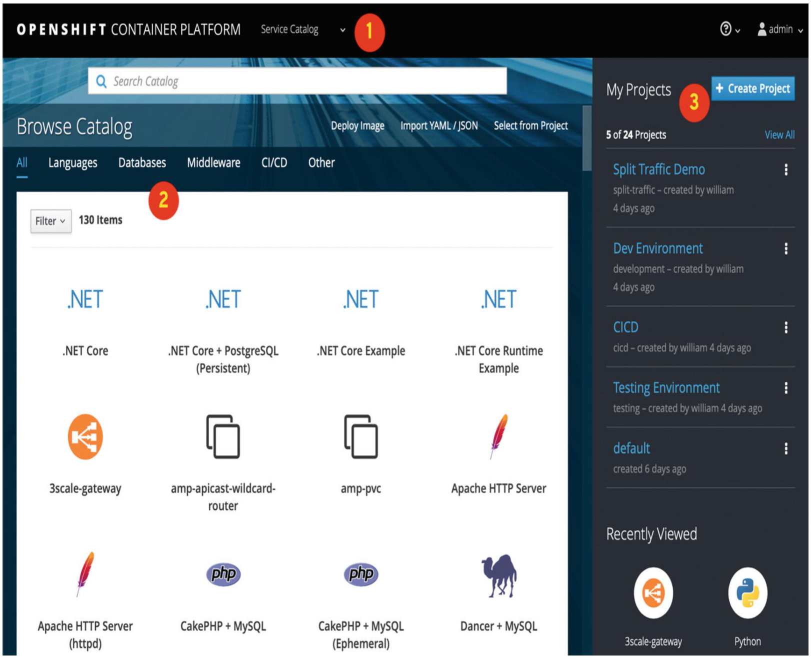


Figure 1-15 The OpenShift self-service portal also known as the developer console

The *Cluster Console* (Figure 1-16), sometimes referred to as the *Cluster Administrator Console,* provides access to cluster operations and functions. At first glance it provides a cluster health and status view (see #1 and #2 of Figure 1-16).

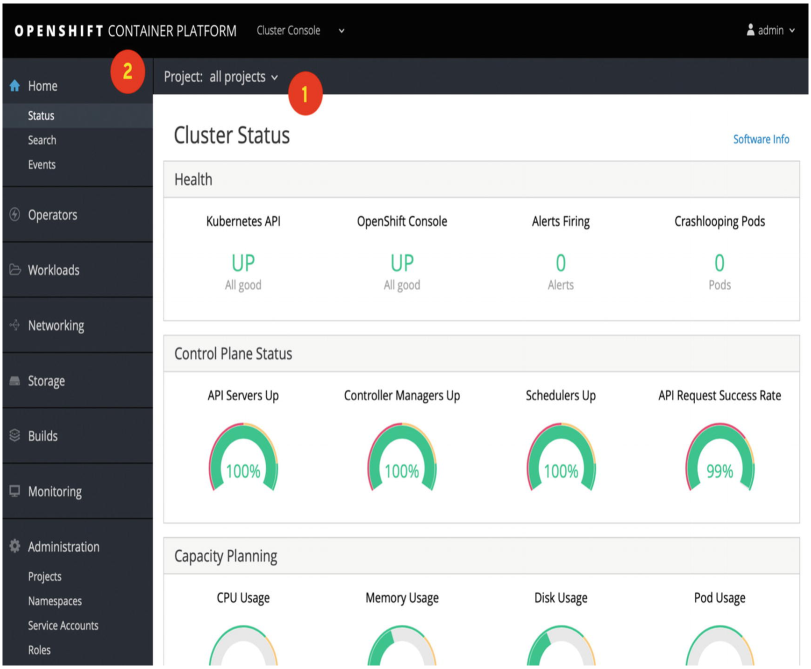
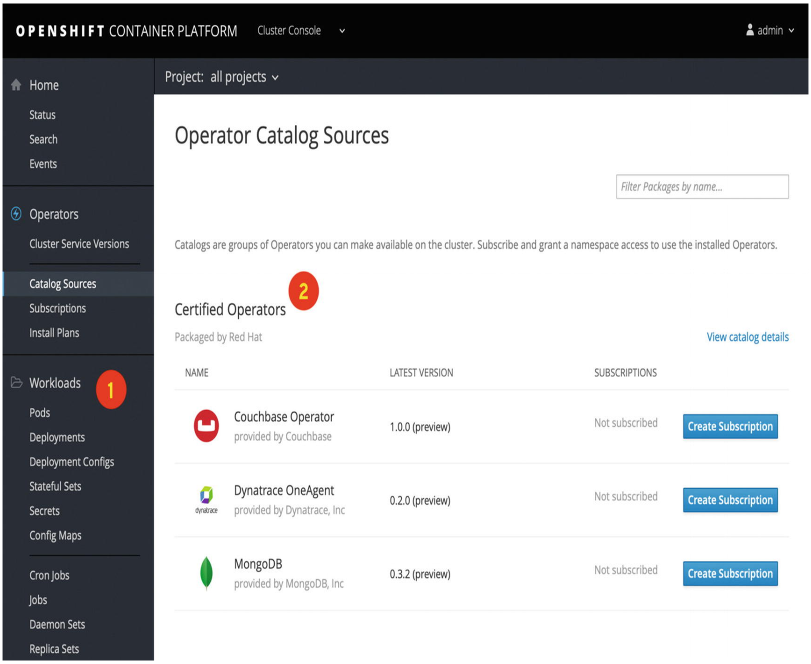


Figure 1-16 The OpenShift Cluster Console also known as the cluster admin console

For users with deep understanding of the Kubernetes, this console also exposes the Kubernetes objects with a more traditional *Container as a Service (CaaS)* experience (see #1 of Figure 1-17). From here, a cluster admin has an aggregated view into the Kubernetes and OpenShift objects like *Namespaces, Pods, Deployments, Secrets, Deployment Configs,* and *ConfigMaps.*



Operator Catalog Sources

Catalogs are groups of Operators you can make available on the cluster. Subscribe and grant a namespace access to use the installed Operators.

Certified Operators

Packaged by Red Hat

NAME	LATEST VERSION	SUBSCRIPTIONS
Couchbase Operator provided by Couchbase	1.0.0 (preview)	Not subscribed
Dynatrace OneAgent provided by Dynatrace, Inc	0.2.0 (preview)	Not subscribed
MongoDB provided by MongoDB, Inc	0.3.2 (preview)	Not subscribed

Figure 1-17 The OpenShift Cluster Console managing subscriptions to Kubernetes Operators

In addition, the *Cluster Console* provides a graphical interface for interacting with *Kubernetes Operators* (see #2 of Figure 1-17).

OpenShift Routers

Steering traffic to applications running on a Kubernetes cluster, until this day with Kubernetes 1.13, it is still highly dependent on where the Kubernetes cluster is running (i.e., on-premise vs. at a Cloud provider). When using a Kubernetes offering from a Cloud provider, they will provide a network service that maps to the *LoadBalancer* object in Kubernetes. Those load balancers provided by the Cloud infrastructure are what is used to steer traffic to the *Service* objects or *Pods* in the cluster.

Outside these options, it is up to the cluster operator to combine Kubernetes constructs with third-party solutions or other Open Source projects to bring the traffic into the cluster. Until now, the options are limited to *NodePort*, *HostPort*, and *Ingress* with an *Ingress Controllers.* The particular implementation details for each one of these objects are beyond the scope of this book, but it's worth having a general overview of these concepts to properly understand the *OpenShift Router*. Figure 1-18 showcases the main difference between using *OpenShift Routes* or *Kubernetes Ingress* and using *NodePorts* or *HostPorts* to steer traffic into the cluster.

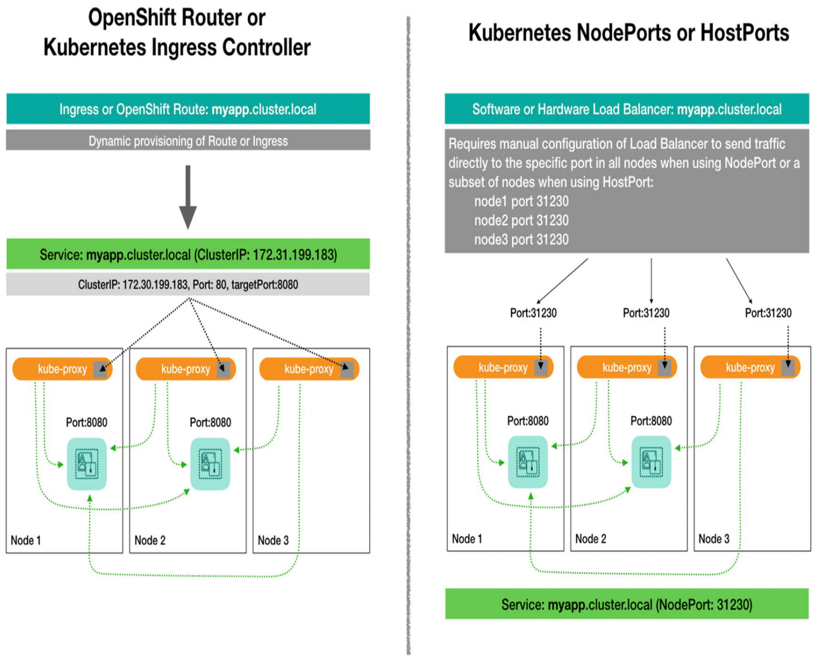

OpenShift Router or Kubernetes Ingress Controller
Ingress or OpenShift Route: myapp.cluster.local
Dynamic provisioning of Route or Ingress
Service: myapp.cluster.local (ClusterIP: 172.31.199.183)
ClusterIP: 172.30.199.183, Port: 80, targetPort:8080
kube-proxy
Port:8080
Node 1
Node 2
Node 3
Kubernetes NodePorts or HostPorts
Software or Hardware Load Balancer: myapp.cluster.local
Requires manual configuration of Load Balancer to send traffic directly to the specific port in all nodes when using NodePort or a subset of nodes when using HostPort:
node1 port 31230
node2 port 31230
node3 port 31230
Port:31230
Service: myapp.cluster.local (NodePort: 31230)

Figure 1-18 OpenShift Routes, Kubernetes Ingress, NodePorts, and HostPorts

NOTE OpenShift supports *OpenShift Routes* and the native Kubernetes *Ingress*, *NodePort*, and *HostPort* resources.

The main difference to keep in mind is that when using *NodePorts*[5] or *HostPorts*,[6] the user is responsible for the configuration and updates to the configuration of the external load balancer or proxy used to steer the traffic toward all the *Nodes* or subset of *Nodes* when using *HostPorts.*

The *OpenShift Router* and the *OpenShift Routes* are a predecessor of the *Ingress Controller* and the *Ingress* object. Even when *Ingress* and *Ingress Controllers* are still available since Kubernetes 1.1, they are still considered Beta in Kubernetes 1.13.[7] There is still no feature parity between *Routes*[8] and *Ingress* objects as it can be seen from Table 1-1.

Table 1-1 OpenShift Routes vs. Kubernetes Ingress [9]

Feature	Ingress	Route
Standard Kubernetes object	X	
External access to services	X	X
Persistent (sticky) sessions	X	X
Load-balancing strategies	X	X
Rate-limit and throttling	X	X
IP whitelisting	X	X
TLS edge termination	X	X
TLS re-encryption		X

Feature	Ingress	Route
TLS passthrough		X
Multiple weighted backends (split traffic)		X
Pattern-based hostname		X
Wildcard domains		X

When using the *Ingress* object in OpenShift, internally, the *Ingress Controller* creates one or more *Route* objects to satisfy the conditions specified by the *Ingress* configuration file. Listing 1-1 represents the *Ingress* configuration file, and Listing 1-2 is the resulting *Route* configuration.

```
kind: Ingress
apiVersion: extensions/v1beta1
metadata:
  name: example
spec:
  rules:
  - host: example.com
    http:
     paths:
     - path: /example
       backend:
        serviceName: example-svc
        servicePort: 80
```

Listing 1-1 Define an Ingress object for example.com

```
kind: Route
apiVersion: route.openshift.io/v1
metadata:
  # Note: The Route name is auto generated by route
object
  # using the Ingress name as prefix
  name: example-a24dc
  ownerReferences:
  - apiVersion: extensions/v1beta1
    kind: Ingress
    name: example
    controller: true
spec:
  host: example.com
  path: /example
  to:
    name: example-svc
  port:
     targetPort: 80
```

Listing 1-2 Resulting Route object for example.com

OpenShift Registry

One of the ancillary services required by Kubernetes is a container registry where the OCI-compliant container runtime can pull the container images. OpenShift provides an integrated container registry known as the *OpenShift Container Registry (OCR)* . This is not a replacement to the organization's enterprise container registries. The purpose of the *OCR* is to provide a built-in location to store images that are deployed into the cluster or images build by the cluster using the native build strategies[10] like Source-to-Image (S2I).[11]

The *OpenShift Container Registry* is hosted on the *Infrastructure Nodes* (refer to Figure 1-13). Among the additional capabilities available with the OCR is the ability to trigger redeployments if a new version of the container image becomes available in the registry.

Summary

In this chapter we provided a map between the Kubernetes architecture and constructs and the OpenShift architecture. We saw how OpenShift is built on top of the Kubernetes primitives and then augment its capabilities by integrating additional Open Source projects. The result is an integrated multitenant Kubernetes platform which enables developers to deploy applications into a Kubernetes cluster without understanding or learning the specifics of Kubernetes while providing the operations teams the ability to manage Kubernetes with a low learning curve. All of this while being a pluggable architecture in which any of the components can be swapped by other projects or software providing the specific capabilities.

Chapter 2 goes into the details on how high availability is achieved for the OpenShift platform and in each one of the core components.

Footnotes

1 In some documentation, you may find a statement about the existence of six namespaces, and in other documentations, you will find seven namespaces listed. Those lists do not count the *cgroup namespace* which virtualizes the *cgroup* capabilities as a namespace. For details about the namespace vs. the capability, refer to the Linux man page *cgroup_namespaces,* the Linux man page for *cgroups,* and the Linux man page for *namespaces.*

2 The CRI specification defines four actions: CreatePod, StartPod, StopPod, and RemovePod.

3 For more details, visit `https://github.com/prometheus/node_exporter`

4 For more details, visit https://github.com/kubernetes/node-problem-detector

5 When using NodePorts, the requested or dynamically assigned port is allocated in all the nodes of the particular cluster. Additional details are available at https://kubernetes.io/docs/concepts/services-networking/service/#nodeport

6 In general the use of *HostPort* is discouraged. Acceptable use cases are DaemonSets or some networking services. Upstream *HostPort* documentation is scarce, but functionality is similar to NodePorts but for a subset of *Nodes*.

7 Kubernetes Ingress feature state: https://kubernetes.io/docs/concepts/services-networking/ingress/#prerequisites

8 OpenShift Routes: https://docs.openshift.com/container-platform/3.11/architecture/networking/routes.html

9 Reference https://blog.openshift.com/kubernetes-ingress-vs-openshift-route/

10 For information about build strategies, visit https://docs.openshift.com/container-platform/3.11/architecture/core_concepts/builds_and_image_streams.htn

11 Source-to-Image (S2I) is an Open Source project (https://github.com/openshift/source-to-image) to create container images from source code. For information on how to use S2I in OpenShift, refer to https://docs.openshift.com/container-platform/3.11/creating_images/s2i.html

PREV
Front Matter

NEXT
2. High Availability