Chapter 8. Container Orchestration with OpenShift and Kubernetes

Abstract

Goal	Orchestrate containerized applications with Kubernetes and OpenShift.
Objectives	• Deploy an application in OpenShift. • Deploy a multi-pod application to OpenShift and make it externally available.
Sections	• Deploy Applications in OpenShift (and Guided Exercise) • Multi-pod Applications (and Guided Exercise)
Lab	• Container Orchestration with Kubernetes and OpenShift

Deploy Applications in OpenShift

Objectives

- Deploy an application in OpenShift.

Kubernetes and Red Hat OpenShift

Kubernetes is an orchestration platform that simplifies the deployment, management, and scaling of containerized applications. Kubernetes uses several servers, called *nodes*, to ensure the resiliency and scalability of the deployed applications. Kubernetes forms a cluster of servers that run containers and are centrally managed by a set of control plane servers. A server can act as both a control plane node and a compute node, but those roles are usually separated for increased stability, security, and manageability.

Red Hat OpenShift is a set of modular components and services that use and expand Kubernetes. Red Hat OpenShift adds capabilities such as remote management, increased security, monitoring and auditing, application lifecycle management, and self-service interfaces for developers.

Based on your infrastructure needs, you can choose from various Red Hat OpenShift editions. Red Hat OpenShift Container Platform (RHOCP) is one of the self-managed options, which you can install and configure on your own infrastructure. RHOCP supports many different target platforms, such as Amazon Web Services (AWS), Microsoft Azure, and bare metal.

Red Hat OpenShift is also available as a cloud managed service. A managed platform can reduce your operational effort and increase your productivity, because you do not need to maintain the infrastructure layer. Examples of Red Hat OpenShift managed services are Microsoft Azure Red Hat OpenShift (ARO) and Red Hat OpenShift Service on AWS (ROSA), among others.

> **NOTE**
>
> The remainder of this chapter covers RHOCP, but the same concepts apply to the managed Red Hat OpenShift editions.

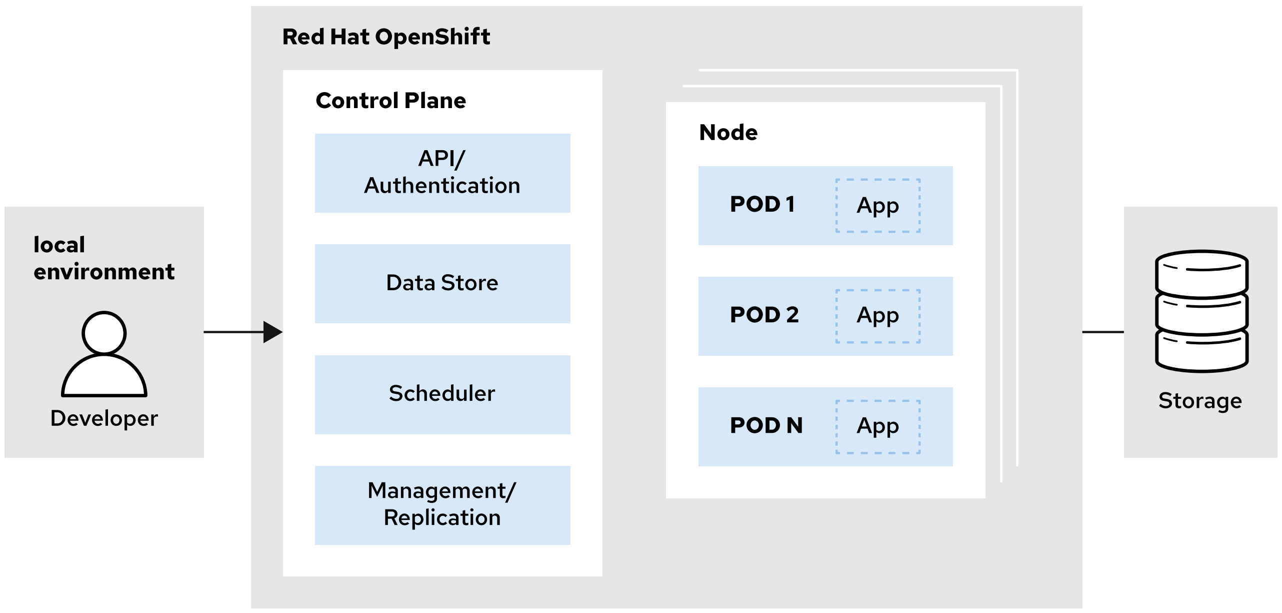


Figure 8.1: RHOCP architecture

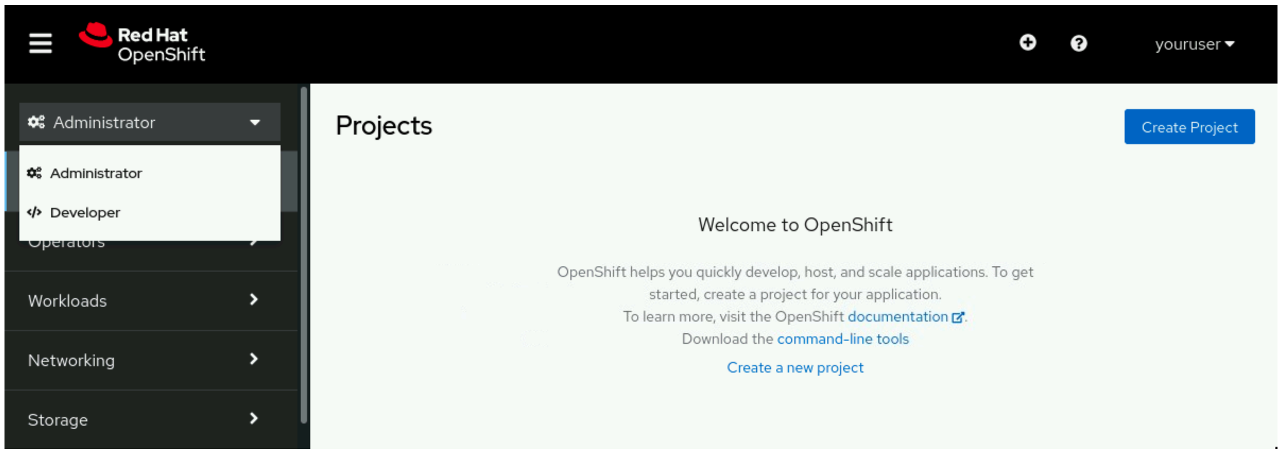
Developers deploy application containers grouped in *pods*. Pods are Kubernetes objects that represent the smallest deployment unit managed by RHOCP.

Developers group related pods and other RHOCP objects in *projects*. A project is an RHOCP resource which is similar to the *namespace* Kubernetes resource. Grouping RHOCP objects in projects is useful, for example, for configuring limits on resource usage for teams or applications.

The RHOCP Web Console

The RHOCP web console is a browser-based user interface that you can use to interact with RHOCP. You can deploy and manage your applications by using the web console.

The web console provides the *Administrator* and *Developer* perspectives. The Developer perspective focuses on the status and management of the deployed applications. The Administrator perspective focuses on the status and management of RHOCP and its resources.



Welcome to OpenShift

OpenShift helps you quickly develop, host, and scale applications. To get started, create a project for your application.
To learn more, visit the OpenShift documentation
Download the command-line tools

Figure 8.2: The Administrator and Developer perspectives

The *Topology* view in the Developer perspective provides a visual representation of deployed applications.

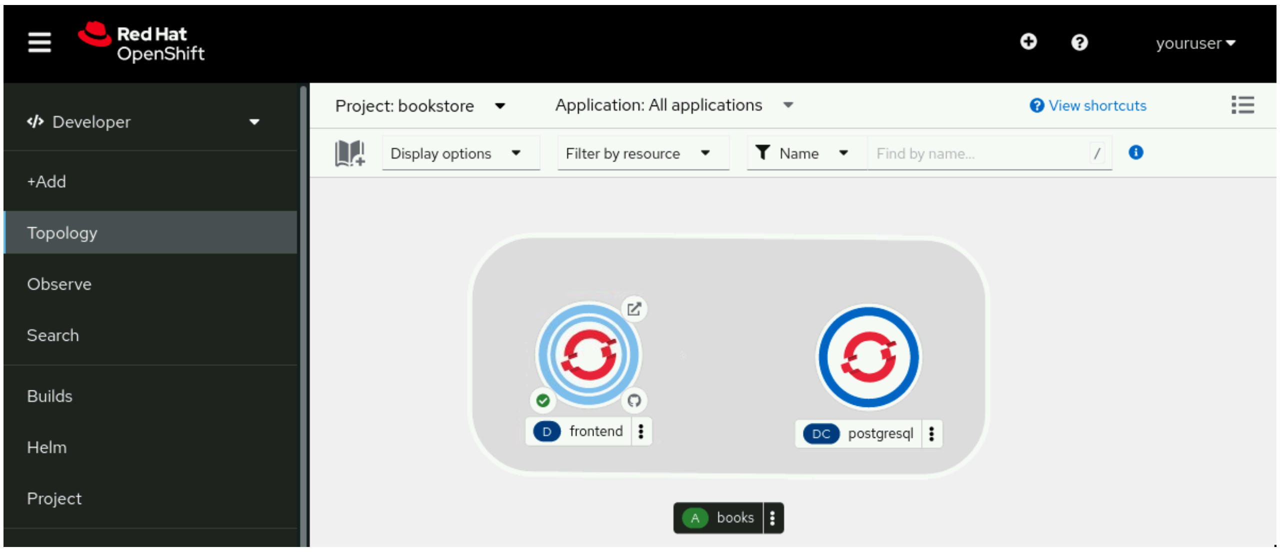


Figure 8.3: The Topology view

The RHOCP CLI

Command-line utilities (CLI) provide an alternative for interacting with your RHOCP cluster. Developers familiar with Kubernetes can use the `kubectl` utility to manage an RHOCP cluster. This course uses the `oc` command-line utility, which is designed to take advantage of additional RHOCP features.

The CLI utilities provide developers with a range of commands that are useful for managing the RHOCP cluster and its applications. Each command is translated into an API call, and the response is displayed in the command line.

The following is a list of the most common `oc` commands.

oc login
Before you can interact with your RHOCP cluster, you must authenticate your requests. Use the `login` command to authenticate your requests.

For example, in this course, you can use the following command:

```
[user@host ~]$ oc login https://api.ocp4.example.com:6443
Username: developer
Password: developer
Login successful.

You don't have any projects. You can try to create a new project, by running

  $ oc new-project <projectname>

Welcome to OpenShift! See 'oc help' to get started.
```

oc get
Use the `get` command to retrieve a list of selected resources in the selected project.

You must specify the resource type to list.

For example, the following command returns the list of the `pod` resources in the current project.

```
[user@host ~]$ oc get pod
NAME                          READY   STATUS    RESTARTS   AGE
quotes-api-6c9f758574-nk8kd   1/1     Running   0          39m
quotes-ui-d7d457674-rbkl7     1/1     Running   0          67s
```

oc create
Use the `create` command to create an RHOCP resource. Developers commonly use the `-f` flag to indicate the file that contains the JSON or YAML representation of an RHOCP resource.

For example, to create resources from the `pod.yaml` file, use the following command:

```
[user@host ~]$ oc create -f pod.yaml
pod/quotes-pod created
```

RHOCP resources in YAML format are discussed later.

oc delete

Use the `delete` command to delete an existing RHOCP resource. You must specify the resource type and the resource name.

For example, to delete the `quotes-ui` pod, use the following command:

```
[user@host ~]$ oc delete pod quotes-ui
pod/quotes-ui deleted
```

oc logs

Use the `logs` command to print the standard output of a pod. This command requires a pod name as an argument. You can print only logs of a container in a pod, which means the resource type is omitted.

For example, to print the logs from the `react-ui` pod, use the following command:

```
[user@host ~]$ oc logs react-ui
Compiled successfully!

You can now view ts-page in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://10.0.1.23:3000
...output omitted...
```

The `oc` utility provides equal functionality to the web console. For a full list of commands, execute `oc --help`. Additionally, you can execute `oc` *`command`* `--help`, for example `oc logs --help` to view the `oc logs` command documentation.

RHOCP Resources

Developers configure RHOCP by using a set of Kubernetes and RHOCP-specific objects. When you create or modify an object, you make a persistent record of the intended state. RHOCP reads the object and modifies the current state accordingly.

NOTE

RHOCP objects generally use the YAML definition format because the YAML format is easy to read and understand.

If you are not familiar with the YAML format, see the references section for more information.

Shared Resource Fields

All RHOCP and Kubernetes objects can be represented as a JSON or YAML structure with common fields. Consider the following pod object in the YAML format:

```
kind: Pod ❶
apiVersion: v1 ❷
metadata: ❸
  name: example-pod
  namespace: example-namespace
spec: ❹
...definition omitted...
status: ❺
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2022-08-19T12:59:22Z"
    status: "True"
    type: PodScheduled
  containerStatuses:
  - containerID: cri-o://e37c....f5c2
    image: quay.io/example/awesome-container:latest
    lastState: {}
    name: podman-quotes-ui
    ready: true
...object omitted...
```

❶ Schema identifier. In this example, the object conforms to the pod schema.

❷ Identifier of the object schema version.

❸ Metadata for a given resource, such as annotations, labels, name, namespace, and others. The `.metadata.namespace` field refers to an RHOCP project.

4 The desired state of the object.

5 Current state of the object. This field is provided by RHOCP, and lists information such as runtime status, readiness, container image, and others.

Every RHOCP resource contains the `.kind`, `.apiVersion`, `.spec` and `.status` fields. However, when you create an object definition, you do not need to provide the `.status` field. The `.status` field is useful, for example, for troubleshooting the reason for a pod scheduling error.

Use the `oc explain` command to get information about valid fields for an object. For example, execute `oc explain pod` to get information about possible Pod object fields. You can use the YAML path to get information about a particular field, for example:

```
[user@host ~]$ oc explain pod.metadata.name
KIND:     Pod
VERSION:  v1

FIELD:    name <string>

DESCRIPTION:
...output omitted...
```

Label Kubernetes Objects

Labels are key-value pairs that you define in the `.metadata.labels` object, for example:

```
kind: Pod
apiVersion: v1
metadata:
  name: example-pod
  labels:
    app: example-pod
    group: developers
...object omitted...
```

The preceding example contains the `app=example-pod` and `group=developers` labels. Developers often use labels to target a set of objects by using the `-l` or the equivalent `--selector` option. For example, the following `oc get` command lists pods that contain the `group=developers` label:

```
[user@host ~]$ oc get pod --selector group=developers
NAME                        READY   STATUS    RESTARTS   AGE
example-pod-6c9f758574-7fhg 1/1     Running   5          11d
```

Deploy Applications as Pods to RHOCP

A pod represents a group of one or multiple containers that share resources, such as a network interface or file system.

Grouping containers into pods is useful for implementing multi-container design patterns, such as using an initialization container to load application data when a pod starts for the first time. Multi-container design patterns are out of scope for this course.

You can use pods locally or on a cluster. On a local system, Podman can manage pods by using the `podman pod` command. This is useful for developing and testing applications that use a multi-container design pattern. On Kubernetes or RHOCP, pods are the atomic unit for deploying and managing your applications. For example, you can deploy a single-container application by creating a pod that contains a single container.

Pods do not expose advanced functionality, such as application scaling or zero-downtime updates. Kubernetes and RHOCP implement such functionality with objects that control pod sets, such as the *Deployment* object. Controller objects are covered in a later section.

Create Pods Declaratively

Storing a pod definition in a file is called the *declarative approach* for creating RHOCP objects, because you declare the intended pod state outside of RHOCP. This means you can store the pod definition in a version control system, such as Git, and incrementally change your definition as your application evolves.

The following YAML object demonstrates the important fields of a pod object:

```
kind: Pod ❶
apiVersion: v1
metadata:
  name: example-pod ❷
  namespace: example-project ❸
spec: ❹
  containers: ❺
  - name: example-container ❻
    image: quay.io/example/awesome-container ❼
    ports: ❽
    - containerPort: 8080
    env: ❾
    - name: GREETING
      value: "Hello from the awesome container"
```

❶ This YAML object defines a pod.

❷ The `.metadata.name` field defines the name of the pod.

❸ The project in which you create the pod. If this project does not exist, then the pod creation fails. If you do not specify a project, then RHOCP uses the currently configured project.

❹ The `.spec` field contains the pod object configuration.

❺ The `.spec.containers` field defines containers in a pod. In this example, the `example-pod` contains one container.

❻ The name of the container inside of a pod. Container names are important for `oc` commands when a pod contains multiple containers.

❼ The image for the container.

❽ The port metadata specifies what ports the container uses. This property is similar to the `EXPOSE` Containerfile property.

❾ The `env` property defines environment variables.

You can create the pod object by saving the YAML definition into a file and then using the `oc` command, for example:

```
[user@host ~]$ oc create -f pod.yaml
pod/example-pod created
```

Create Pods Imperatively

RHOCP also provides the *imperative approach* to create RHOCP objects. The imperative approach uses the `oc run` command to create a pod without a definition.

The following command creates the `example-pod` pod:

```
[user@host ~]$ oc run example-pod \ ❶
  --image=quay.io/example/awesome-container \ ❷
  --env GREETING='Hello from the awesome container' \ ❸
  --port=8080 ❹
pod/example-pod created
```

❶ The pod `.metadata.name` definition.

❷ The image used for the single container in this pod.

❸ The `--env` option injects an environment variable in the container of this pod.

❹ The port metadata definition.

The imperative commands are a faster way of creating pods, because such commands do not require a pod object definition. However, developers lose the ability to version and incrementally change the pod definition.

Generally, developers test the deployment by using imperative commands, and then use the imperative commands to generate the pod object definition. Use the `--dry-run=client` option to avoid creating the object in RHOCP. Additionally, use the `-o yaml` or `-o json` option to configure the definition format.

The following command is an example of generating the YAML definition for the `example-pod` pod.

```
[user@host ~]$ oc run example-pod \
  --image=quay.io/example/awesome-container \
  --env GREETING='Hello from the awesome container' \
  --port=8080 \
  --dry-run=client -o yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: example-pod
  name: example-pod
spec:
  containers:
...output omitted...
```

Application Networking in RHOCP

Developers configure internal pod-to-pod network communication in RHOCP by using the *Service* object. Applications send requests to the service name and port. RHOCP provides a virtual network which reroutes such requests to the pods that the service targets by using labels.

Create Services Declaratively

The following YAML object demonstrates a service object:

```
apiVersion: v1
kind: Service
metadata:
  name: backend
spec:
  ports:
  - port: 8080 ❶
    protocol: TCP
    targetPort: 8080 ❷
  selector: ❸
    app: backend-app
```

❶ Service port. This is the port on which the service listens.

❷ Target port. This is the pod port to which the service routes requests. This port corresponds to the `containerPort` value in the pod definition.

❸ The selector configures which pods to target. In this case, the service routes to any pods that contain the `app=backend-app` label.

The preceding example defines the `backend` service. This means that applications can send requests to the `http://backend:8080` URL. Such requests are routed to the pods with the `app=backend-app` label on port `8080`.

In RHOCP, the default service type is `ClusterIP`, which means the service is used for pod-to-pod routing within the RHOCP cluster. Other service types, such as the `LoadBalancer` service, are outside of the scope of this course.

Create Services Imperatively

Similarly to pods, you can create services imperatively by using the `oc expose` command. The following example creates a service for the `backend-app` pod:

```
[user@host ~]$ oc expose pod backend-app \
  --port=8080 \ ❶
  --targetPort=8080 \ ❷
  --name=backend-app ❸
service/backend-app exposed
```

❶ Port on which the service listens.

❷ Target container port.

❸ Service name.

You can also use the `--dry-run=client` and `-o` options to generate a service definition, for example:

```
[user@host ~]$ oc expose pod backend-app \
  --port=8080 \
  --targetPort=8080 \
  --name=backend-app \
  --dry-run=client -o yaml
apiVersion: v1
kind: Service
metadata:
  name: backend-app
spec:
...output omitted...
```

REFERENCES

For more information about the YAML format, refer to the *YAML in a Nutshell* chapter in the Red Hat Enterprise Linux Atomic Host 7 *Getting Started with Kubernetes* documentation at https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html-single/getting_started_with_kubernetes/index#yaml_in_a_nutshell

For more information about the architecture, refer to the *Architecture overview* chapter in the Red Hat OpenShift Container Platform 4.18 *Architecture* documentation at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.18/html-single/architecture/index#architecture-overview

For more information about the web console, refer to the Red Hat OpenShift Container Platform 4.18 *Web console* documentation at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.18/html-single/web_console/index

For more information about the `oc` CLI, refer to the *OpenShift CLI* chapter in the Red Hat OpenShift Container Platform 4.18 *CLI tools* documentation at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.18/html-single/cli_tools/index#openshift-cli-oc

For more information about pods, refer to the *Using Pods* section in the *Overview of Nodes* chapter in the Red Hat Openshift Container Platform 4.18 *Nodes* documentation at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.18/html-single/nodes/index#working-with-pods

For more information about Red Hat OpenShift Service on AWS, refer to the Red Hat OpenShift Service on AWS 4 *Introduction to ROSA* documentation at https://access.redhat.com/documentation/en-us/red_hat_openshift_service_on_aws/4/html/introduction_to_rosa/index

Microsoft Azure Red Hat OpenShift

Kubernetes API Conventions

Kubernetes - Service

Service API Reference

Pod API Reference