Introducing Red Hat OpenShift Container Platform

Overview	
Goal	List the features and describe the architecture of the OpenShift Container Platform.
Objectives	● Describe the typical use of the product and list its features. ● Describe the architecture of OpenShift.

Describing OpenShift Container Platform Features

Features of Red Hat OpenShift Container Platform

Red Hat OpenShift Container Platform (*OpenShift*) is a container application platform that provides developers and IT organizations with a cloud application platform for deploying new applications on secure, scalable resources with minimal configuration and management overhead.

Built on Red Hat Enterprise Linux, Docker, and Kubernetes, OpenShift provides a secure and scalable multitenant operating system for today's enterprise applications, while providing integrated application runtimes and libraries. OpenShift brings a robust, flexible, and scalable container platform to customer data centers, enabling organizations to implement a platform that meets security, privacy, compliance, and governance requirements.

Customers who prefer not to manage their own OpenShift clusters can use Red Hat OpenShift Online, a public cloud platform provided by Red Hat. Both OpenShift Container Platform and OpenShift Online are based on the OpenShift Origin open

source software project, which itself builds on a number of other open source projects such as Docker and Kubernetes.

Applications run as *containers* , which are isolated partitions inside a single operating system. Containers provide many of the same benefits as virtual machines, such as security, storage, and network isolation, while requiring far fewer hardware resources and being quicker to launch and terminate. The use of containers by OpenShift helps with the efficiency, elasticity, and portability of the platform itself as well as its hosted applications.

The main features of OpenShift are listed below:

- **Self-service platform** : OpenShift allows developers to create applications from templates or from their own source code management repositories using Source-to-Image (S2I). System administrators can define resource quotas and limits for users and projects to control the use of system resources.
- **Multilingual support** : OpenShift supports Java, Node.js, PHP, Perl, and Ruby directly from Red Hat, and many others from partners and the larger Docker community. MySQL, PostgreSQL, and MongoDB databases directly from Red Hat and others from partners and the Docker community. Red Hat also supports middleware products such as Apache httpd, Apache Tomcat, JBoss EAP, ActiveMQ, and Fuse running natively on OpenShift.
- **Automation** : OpenShift provides application life-cycle management features to automatically rebuild and redeploy containers when upstream source or container images change. Scale out and fail over applications based on scheduling and policy. Combine composite applications built from independent components or services.
- **User interfaces** : OpenShift provides a web UI for deploying and monitoring applications, and a CLI for remote management of applications and resources. It supports the Eclipse IDE and JBoss Developer Studio plug-ins so that developers can stay with familiar tools, and a REST API for integration with third-party or in-house tools.
- **Collaboration** : OpenShift allows you to share projects and customized runtimes within an organization or with the larger community.
- **Scalability and High Availability** : OpenShift provides container multitenancy and a distributed application platform including elasticity to handle increased traffic on demand. It provides high availability so that applications can survive events such as the loss of a physical machine. OpenShift provides automatic discovery of container health and automatic redeployment.

- **Container portability** : In OpenShift, applications and services are packaged using standard container images and composite applications are managed using Kubernetes. These images can be deployed to other platforms built on those base technologies.
- **Open source** : No vendor lock-in.
- **Security** : OpenShift provides multilayered security using SELinux, role-based access control, and the ability to integrate with external authentication systems such as LDAP and OAuth.
- **Dynamic Storage Management** : OpenShift provides both static and dynamic storage management for container data using the Kubernetes concepts of Persistent Volumes and Persistent Volume Claims.
- **Choice of cloud** (or no cloud): Deploy OpenShift Container Platform on bare-metal servers, hypervisors from multiple vendors, and most IaaS cloud providers.
- **Enterprise Grade** : Red Hat provides support for OpenShift, selected container images, and application runtimes. Trusted third-party container images, runtimes, and applications are certified by Red Hat. You can run in-house or third-party applications in a hardened, secure environment with high availability provided by OpenShift.
- **Log Aggregation and Metrics** : Logging information from applications deployed on OpenShift can be collected, aggregated, and analyzed in a central location. OpenShift enables you to collect metrics and runtime information in real time about your applications and helps optimize the performance continuously.

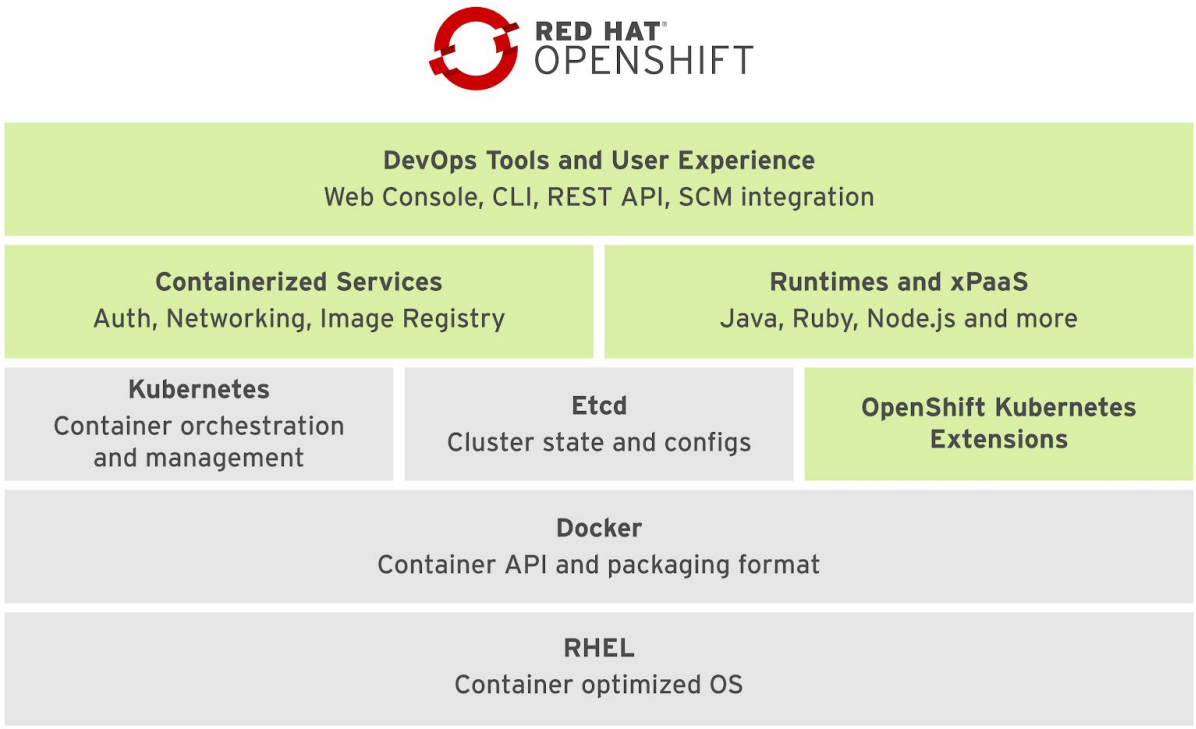
OpenShift is an enabler for microservice architectures, while also supporting more traditional workloads. Many organizations will also find OpenShift native features sufficient to enable a DevOps process, while others will find it is easy to integrate with both standard and custom Continuous Integration/Continuous Deployment tools.

Describing the OpenShift Container Platform Architecture

Overview of OpenShift Container Platform Architecture

OpenShift Container Platform is a set of modular components and services built on top of Red Hat Enterprise Linux, Docker, and Kubernetes. OpenShift adds capabilities such as remote management, multitenancy, increased security, application life-cycle management, and

self-service interfaces for developers. The following figure illustrates the OpenShift software stack:



In the above figure, going from bottom to top, and from left to right, the basic container infrastructure is shown, integrated and enhanced by Red Hat:

- The base OS is Red Hat Enterprise Linux (RHEL).
- **Docker** provides the basic container management API and the container image file format.
- **Kubernetes** manages a cluster of hosts (physical or virtual) that run containers. It works with resources that describe multicontainer applications composed of multiple resources, and how they interconnect.
- **Etcd** is a distributed key-value store, used by Kubernetes to store configuration and state information about the containers and other resources inside the OpenShift cluster.

OpenShift adds to the Docker + Kubernetes infrastructure the capabilities required to provide a container application platform. Continuing from bottom to top and from left to right:

- **OpenShift-Kubernetes extensions** are additional resource types stored in Etcd and managed by Kubernetes. These additional resource types form the

OpenShift internal state and configuration, alongside application resources managed by standard Kubernetes resources.

- **Containerized services** fulfill many infrastructure functions, such as networking and authorization. Some of them run all the time, while others are started on demand. OpenShift uses the basic container infrastructure from Docker and Kubernetes for most internal functions. That is, most OpenShift internal services run as containers managed by Kubernetes.
- **Runtimes and xPaaS** are base container images ready for use by developers, each preconfigured with a particular runtime language or database. They can be used as-is or extended to add different frameworks, libraries, and even other middleware products. The *xPaaS* offering is a set of base images for JBoss middleware products such as JBoss EAP and ActiveMQ.
- **DevOps tools and user experience** : OpenShift provides a Web UI and CLI management tools for developers and system administrators, allowing the configuration and monitoring of applications and OpenShift services and resources. Both Web and CLI tools are built from the same REST APIs, which can be leveraged by external tools like IDEs and CI platforms. OpenShift can also reach external SCM repositories and container image registries and bring their artifacts into the OpenShift cloud.

OpenShift does not hide the core Docker and Kubernetes infrastructure from developers and system administrators. Instead it uses them for its internal services, and allows importing raw containers and Kubernetes resources into the OpenShift cluster so that they can benefit from added capabilities. The reverse is also true: Raw containers and resources can be exported from the OpenShift cluster and imported into other Docker-based infrastructures.

The main value that OpenShift adds to Docker + Kubernetes is automated development workflows, so that application building and deployment happen inside the OpenShift cluster, following standard processes. Developers do not need to know the low-level Docker details. OpenShift takes the application, packages it, and starts it as a container.

Master and Nodes

An OpenShift cluster is a set of *node* servers that run containers and are centrally managed by a set of *master* servers. A server can act as both a master and a node, but those roles are usually segregated for increased stability.

While the OpenShift software stack presents a static perspective of software packages that form OpenShift, the following figure presents a dynamic view of how OpenShift works:

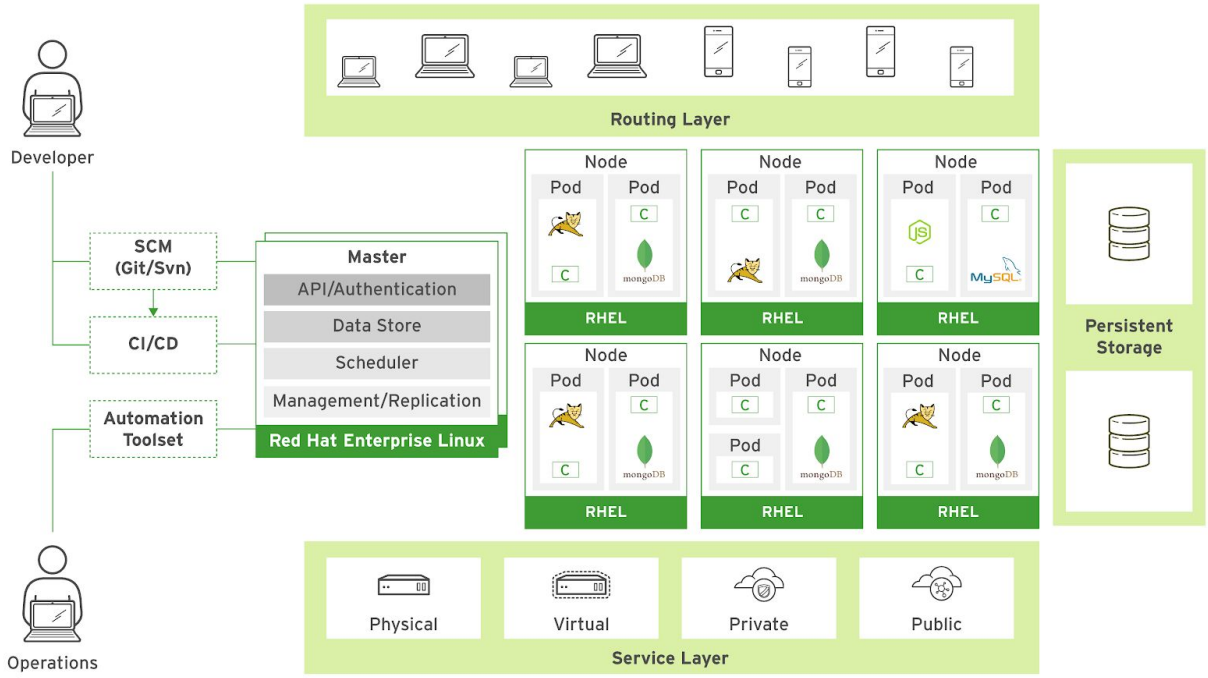


Figure 1.2: OpenShift Container Platform master and nodes

The master runs OpenShift core services such as authentication, and provides the API entry poFigure 1.2: OpenShift Container Platform master and nodes

The master runs OpenShift core services such as authentication, and provides the API entry point for administration. The nodes run applications inside containers, which are in turn grouped into pods. This division of labor actually comes from Kubernetes, which uses the term *minions* for nodes.

OpenShift masters run the`Kubernetes master`services and`Etcd`daemons, while the nodes run the Kubernetes`kubelet`and`kube-proxy`daemons. While not shown in the figure, the masters are also nodes themselves. Scheduler and Management/Replication in the figure are Kubernetes master services, while`Data Store`is the Etcd daemon.

The Kubernetes scheduling unit is the *pod* , which is a grouping of containers sharing a virtual network device, internal IP address, TCP/UDP ports, and persistent storage. A pod can be anything from a complete enterprise application, including each of its layers as a distinct container, to a single microservice inside a single container. For example, a

pod with one container running PHP under Apache and another container running MySQL.

Kubernetes manages **replicas** to scale pods. A replica is a set of pods sharing the same definition. For example, a replica consisting of many Apache and PHP pods running the same container image could be used for horizontally scaling a web application.

int for administration. The nodes run applications inside containers, which are in turn grouped into pods. This division of labor actually comes from Kubernetes, which uses the term *minions* for nodes.

OpenShift masters run the`Kubernetes master`services and`Etcd`daemons, while the nodes run the Kubernetes`kubelet`and`kube-proxy`daemons. While not shown in the figure, the masters are also nodes themselves. Scheduler and Management/Replication in the figure are Kubernetes master services, while`Data Store`is the Etcd daemon.

The Kubernetes scheduling unit is the *pod* , which is a grouping of containers sharing a virtual network device, internal IP address, TCP/UDP ports, and persistent storage. A pod can be anything from a complete enterprise application, including each of its layers as a distinct container, to a single microservice inside a single container. For example, a pod with one container running PHP under Apache and another container running MySQL.

Kubernetes manages **replicas** to scale pods. A replica is a set of pods sharing the same definition. For example, a replica consisting of many Apache and PHP pods running the same container image could be used for horizontally scaling a web application.

OpenShift Projects and Applications

Apart from Kubernetes resources, such as pods and services, OpenShift manages *projects* and *users* . A project groups Kubernetes resources so that access rights can be assigned to users. A project can also be assigned a *quota* , which limits its number of defined pods, volumes, services, and other resources.

There is no concept of an application in OpenShift. The OpenShift client provides a **new-app** command. This command creates resources inside a project, but none of them are application resources. This command is a shortcut to configure a project with common resources for a standard developer workflow. OpenShift uses *labels* to

categorize resources within the cluster. By default, OpenShift uses the *app* label to group related resources together into an *application* .

Building Images with Source-to-Image

Developers and system administrators can use ordinary Docker and Kubernetes workflows with OpenShift, but this requires them to know how to build container image files, work with registries, and other low-level functions. OpenShift allows developers to work with standard source control management (SCM) repositories and integrated development environments (IDEs).

The Source-to-Image (S2I) process in OpenShift pulls code from an SCM repository, automatically detects which kind of runtime that source code needs, and starts a pod from a base image specific to that kind of runtime. Inside this pod, OpenShift builds the application the same way that the developer would (for example, running`maven`for a Java application). If the build is successful, another image is created, layering the application binaries over its runtime, and this image is pushed to an image registry internal to OpenShift. A new pod can then be created from this image, running the application. S2I can be viewed as a complete CI/CD pipeline already built into OpenShift.

There are many variations to CI/CD pipelines, and the pipeline resources are exposed inside the project so they can be tuned to a developer's needs. For example, an external CI tool such as Jenkins could be used to start the build and run tests, then label the newly built image as a success or a failure, promoting it to QA or production. Over time, an organization can create their own templates for those pipelines, including custom builders and deployers.

Managing OpenShift Resources

OpenShift resources, such as images, containers, pods, services, builders, templates, and others, are stored on Etcd and can be managed by the OpenShift CLI, the web console, or the REST API. These resources can be viewed as JSON or YAML text files, and shared on an SCM system like Git or Subversion. OpenShift can even retrieve these resource definitions directly from an external SCM.

Most OpenShift operations are not imperative. OpenShift commands and API calls do not require that an action be performed immediately. OpenShift commands and APIs usually create or modify a resource description stored in Etcd. Etcd then notifies

OpenShift controllers, which warn those resources about the change. Those controllers take action so that the cloud state eventually reflects the change.

For example, if a new pod resource is created, Kubernetes schedules and starts that pod on a node, using the pod resource to determine which image to use, which ports to expose, and so on. As a second example, if a template is changed so that it specifies that there should be more pods to handle the load, OpenShift schedules additional pods (replicas) to satisfy the updated template definition.

OpenShift Networking

Docker networking is very simple. Docker creates a virtual kernel bridge and connects each container network interface to it. Docker itself does not provide a way to allow a pod on one host to connect to a pod on another host. Neither does Docker provide a way to assign a public fixed IP address to an application so that external users can access it.

Kubernetes provides service and route resources to manage network visibility between pods and route traffic from the external world to the pods. A *service* load-balances received network requests among its pods, while providing a single internal IP address for all clients of the service (which usually are other pods). Containers and pods do not need to know where other pods are, they just connect to the service. A *route* provides a fixed unique DNS name for a service, making it visible to clients outside the OpenShift cluster.

Kubernetes service and route resources need external help to perform their functions. A service needs software-defined networking (SDN) which will provide visibility between pods on different hosts, and a route needs something that forwards or redirects packets from external clients to the service internal IP. OpenShift provides an SDN based on **Open vSwitch** , and routing is provided by a distributed **HAProxy** farm.

Persistent Storage

Pods might be stopped on one node and restarted on another node at any time. Consequently, plain Docker storage is inadequate because of its default ephemeral nature. If a database pod was stopped and restarted on another node, any stored data would be lost.

Kubernetes provides a framework for managing external persistent storage for containers. Kubernetes recognizes a *PersistentVolume* resource, which can define

either local or network storage. A pod resource can reference a *PersistentVolumeClaim* resource in order to access storage of a certain size from a PersistentVolume.

Kubernetes also specifies if a PersistentVolume resource can be shared between pods or if each pod needs its own PersistentVolume with exclusive access. When a pod moves to another node, it stays connected to the same PersistentVolumeClaim and PersistentVolume instances. This means that a pod's persistent storage data follows it, regardless of the node where it is scheduled to run.

OpenShift adds to Kubernetes a number of *VolumeProviders* , which provide access to enterprise storage, such as iSCSI, Fibre Channel, Gluster, or a cloud block volume service such as OpenStack Cinder.

OpenShift also provides dynamic provisioning of storage for applications via the *StorageClass* resource. Using dynamic storage, you can select different types of back-end storage. The back-end storage is segregated into different "tiers" depending on the needs of your application. For example, a cluster administrator can define a StorageClass with the name of “fast,” which makes use of higher quality back-end storage, and another StorageClass called “slow,” which provides commodity-grade storage. When requesting storage, an end user can specify a`PersistentVolumeClaim`with an annotation that specifies the value of the StorageClass they prefer.

OpenShift High Availability

High Availability (HA) on an OpenShift Container Platform cluster has two distinct aspects: HA for the OpenShift infrastructure itself (that is, the masters); and HA for the applications running inside the OpenShift cluster.

OpenShift provides a fully supported native HA mechanism for masters by default.

For applications, or "pods," OpenShift handles this by default. If a pod is lost for any reason, Kubernetes schedules another copy, connects it to the service layer and to the persistent storage. If an entire node is lost, Kubernetes schedules replacements for all its pods, and eventually all applications will be available again. The applications inside the pods are responsible for their own state, so they need to maintain application state on their own (for example, by employing proven techniques such as HTTP session replication or database replication).

Image Streams

To create a new application in OpenShift, in addition to the application source code, a base image (the S2I builder image) is required. If either of these two components are updated, a new container image is created. Pods created using the older container image are replaced by pods using the new image.

While it is obvious that the container image needs to be updated when application code changes, it may not be obvious that the deployed pods also need to be updated if the builder image changes.

An *image stream* comprises any number of container images identified by *tags* . It presents a single virtual view of related images. Applications are built against image streams. Image streams can be used to automatically perform an action when new images are created. Builds and deployments can watch an image stream to receive notifications when new images are added, and react by performing a build or deployment, respectively. OpenShift provides several image streams by default, encompassing many popular language runtimes and frameworks.

An *image stream tag* is an alias pointing to an image in an image stream. It is often abbreviated to *istag* . It contains a history of images represented as a stack of all images that the tag ever pointed to. Whenever a new or existing image is tagged with a particular istag, it is placed at the first position (tagged as *latest*) in the history stack. Images previously tagged as *latest* will be available at the second position. This allows for easy rollbacks to make tags point to older images again.

Installing OpenShift Container Platform

Goal	Install OpenShift and configure the cluster.
Objectives	● Prepare the servers for installation. ● Execute the installation steps to build and configure an OpenShift cluster. ● Execute postinstallation tasks and verify the cluster configuration.

General Installation Overview

Red Hat OpenShift Container Platform is delivered as a mixture of RPM packages and container images. The RPM packages are downloaded from standard Red Hat repositories (that is, Yum repositories) using subscription manager, and the container images come from the Red Hat private container registry.

OpenShift Container Platform installations require multiple servers, and these can be any combination of physical and virtual machines. Some of them are *masters* , others are *nodes* , and each type needs different packages and configurations. Red Hat offers two different methods for installing Red Hat OpenShift Container Platform. The first method, know as Quick Installation, can be used for simple cluster setups. Quick Installation uses answers to a small set of questions to bootstrap installation. The second method, known as Advanced Installation, is designed for more complex installations. Advanced Installation uses *Ansible Playbooks* to automate the process.

Beginning with OpenShift Container Platform 3.9, the Quick Installation method is deprecated. As a result, this course uses the Advanced Installation method for installing Red Hat OpenShift Container Platform.

You must prepare cluster hosts prior to initiating installation. After running the Advanced Installation, you perform a smoke test to verify the cluster's functionality. In the comprehensive review lab at the end of the class, you practice manually performing these tasks. A brief introduction of Ansible follows because the Advanced Installation method relies on Ansible.

What is Ansible?

Ansible is an open source automation platform which is used to customize and configure multiple servers in a consistent manner. Ansible Playbooks are used to declare the desired configuration of servers. Any server that is already in the declared state is left unchanged. The configuration of all other servers is adjusted to match the declared configuration. As a result, Ansible Playbooks are describe as *idempotent* . An idempotent playbook can be executed repeatedly and result in the same final configuration.

OpenShift uses Ansible Playbooks and roles for installation, which are included in the atomic-openshift-utils package. The playbooks support a variety of installation and configuration scenarios.

For the purpose of this course, Ansible is responsible for:

- Preparing the hosts for OpenShift installation, such as package installation, disabling services, and configuring the Docker service.

- Installing Red Hat OpenShift.
- Executing smoke tests to verify the Red Hat OpenShift installation.

Installing Ansible

Use the **subscription-manager** command to enable the rhel-7-server-ansible-2.4-rpms repository:

```
#subscription-manager repos --enable="rhel-7-server-ansible-2.4-rpms"
```

When the repository is enabled, use the **yum install** command to install Ansible:

```
#yum install ansible
```

Ansible Playbooks Overview

Ansible Playbooks are used to automate configuration tasks. An Ansible Playbook is a YAML file that defines a list of one or more *plays* . Each play defines a set of tasks that are executed on a specified group of hosts. Tasks can be explicitly defined in the`tasks`section of a play. Tasks can also be encapsulated in an Ansible *role* . The following is an example of an Ansible Playbook:

```
---

- name: Install a File
  hosts: workstations
  vars:
    sample_content: "Hello World!"
  tasks:
    - name: "Copy a sample file to each workstation."
      copy:
        content: "{{ sample_content }}"
        dest: /tmp/sample.txt

- name: Hello OpenShift Enterprise v3.x
  hosts: OSEv3
  roles:
    - hello
```

Ansible Inventory Files

Ansible Playbooks execute against a set of servers. Ansible inventory files define the groups of machines referenced in a set of playbooks. Any host group referenced in a playbook play must have a corresponding entry in the inventory file. For example, OpenShift playbooks operate on, among others, a *masters* host group. An inventory file used with OpenShift Playbooks must define which hosts belong to the`masters`group.

In this class, playbook variables are defined in the inventory file. These variables can be defined for an entire group or can be defined on a per-host basis.

Ansible inventory files are `INI` files. The following inventory file describes the host groups in the classroom environment. This inventory file also includes variable definitions needed for the previous playbook:

```
[workstations]
workstation.lab.example.com

[nfs]
services.lab.example.com

[masters]
master.lab.example.com

[etcd]
master.lab.example.com

[nodes]
master.lab.example.com  hello_message="I am an OSEv3 master."
node1.lab.example.com
node2.lab.example.com

[OSEv3:children]
masters
etcd
nodes
nfs

[OSEv3:vars]
hello_message="I am an OSEv3 machine."

[workstations:vars]
sample_content="This is a workstation machine."
```

Executing Ansible Playbooks

In this class, related playbook artifacts are kept in a project directory. A simple playbook project contains the following artifacts:

- An `inventory` file.
- A `playbooks` directory. The directory may be absent if only a small number of playbooks exist.

- An `ansible.cfg` file. This file customizes the behavior of Ansible command-line utilities. This file often defines the SSH configuration for Ansible commands and specifies the default inventory file for playbooks.

In this class, the `ansible.cfg` file contains the following:

```
[defaults]
remote_user = student
inventory = ./inventory
roles_path = /home/student/do280-ansible/roles
log_path = ./ansible.log

[privilege_escalation]
become = yes
become_user = root
become_method = sudo
```

This project structure simplifies the execution of playbook. From a terminal, change to the project's root directory. Use the **ansible-playbook** command to execute a project playbook:

```
[student@workstation project-dir]$ansible-playbook <playbook-filename>
```

To execute a playbook against a different inventory file use the `-i` option:

```
[student@workstation project-dir]$ansible-playbook
-i<inventory-file><playbook-filename>
```

Preparing the Environment

Ensure the following prerequisites are satisfied before you begin the installation:

- Passwordless SSH is configured for a user account on all remote machines. If the remote user is not the root user, passwordless sudo rights must be granted to the remote user. In the classroom environment, the `student` user account satisfies these requirements.
- The master and node hosts need to be able to communicate with each other. The **ping** command provides a way to verify communication between the master and node hosts.
  ```
  [root@master ~]#ping node1.lab.example.com
  ```
- A wildcard DNS zone must resolve to the IP address of the node running the OpenShift router component. Use the **ping** or **dig** command to call a host name that does not exist in the domain. For example, in the classroom, the wildcard

domain`apps.lab.example.com`is used for all applications running on OpenShift:
`[root@master ~]#`**`dig test.apps.lab.example.com`**

A system administrator with RHCSA or equivalent skills should be able to configure the servers and ensure that they meet most of the above requirements. For the wildcard DNS zone, they might require help from an experienced system administrator with advanced knowledge of DNS server administration.

The OpenShift router needs the wildcard DNS zone. Because the OpenShift router runs on an OpenShift node, system administrators need to plan in advance to configure the DNS with the IP address of the correct node.

The OpenShift Advanced Installation method has additional prerequisites. An Ansible Playbook has been developed for the classroom environment to satisfy these requirements. Run this playbook in order to satisfy the prerequisites listed below:

- Each OpenShift Container Platform cluster machine is a Red Hat Enterprise Linux 7.3, 7.4, or 7.5 host.
- Each OpenShift cluster host (that is, masters and nodes) is registered using Red Hat Subscription Management (RHSM), not RHN Classic. To register hosts, use the **subscription-manager register** command.
- Each host is attached to valid OpenShift Container Platform subscriptions. To attach hosts to a subscription, use the **subscription-manager attach** command.
- Only the required repositories are enabled. Besides the standard RHEL repository (rhel-7-server-rpms), the rhel-7-server-extras-rpms , rhel-7-fast-datapath-rpms , and rhel-7-server-ansible-2.4-rpms repositories are enabled. The rhel-7-server-ose-3.9-rpms repository provides the necessary OpenShift Container Platform packages. To enable the required repositories, use the **subscription-manager repos --enable** command. Enable these repositories on all masters and nodes in the OpenShift cluster.
- Base packages are installed on all OpenShift hosts: wget , git , net-tools , bind-utils , yum-utils , iptables-services , bridge-utils , bash-completion , kexec-tools , sos , psacct , and atomic-openshift-utils . The Advanced Installation method uses playbooks and other installation utilities found in the atomic-openshift-utils package.
- docker is installed and configured on each OpenShift host. By default, the Docker daemon stores container images using a thin pool on a loopback device. For production Red Hat OpenShift clusters, the Docker daemon must use a thin pool logical volume. Use the **docker-storage-setup** command to set up appropriate

storage for the Docker daemon. The Red Hat OpenShift Documentation covers many of the considerations for setting up Docker storage on OpenShift hosts.

Steps for Installation:

Copy all the ISO's in your YUM SERVER and mount them to prepare it as an FTP server.

Networking:

1. MASTER, NODE1, NODE2, WORKSTATION, REGISTRY:
   vim /etc/sysconfig/network-scripts/ifcfg-enp0s3

   BOOTPROTO= static
   ONBOOT=yes
   IPADDR= (ifconfig)
   NETMASK= ifconfig
   GATEWAY= ([root@node1 ~]# route -n)
   DNS1=(ip of dns server)

   Systemctl restart NetworkManager
   Set hostname

2. DNS(YUM SERVER)
   systemctl stop firewalld
   systemctl disable firewalld
   Yum install dnsmasq -y
   vim /etc/hosts

   192.168.225.108 master.example.com master
   192.168.225.237 node1.example.com node1
   192.168.225.130 node2.example.com node2
   192.168.225.178 registry.example.com registry
   192.168.225.167 workstation.example.com workstation
   ---
   systemctl restart dnsmasq
   systemctl enable dnsmasq
   vim /etc/dnsmasq.conf

   Line 46:
   resolv-file=/etc/realdns

Vim /etc/realdns

Nameserver 8.8.8.8
---
systemctl restart dnsmasq

AT WORKSTATION:

[root@workstation ~]# ssh-key

[root@workstation ~]# ssh-copy-id root@master

[root@workstation ~]# ssh-copy-id root@node1

[root@workstation ~]# ssh-copy-id root@node2

[root@workstation ~]# ssh-copy-id root@registry

[root@workstation ~]# ssh-copy-id root@workstation

[root@workstation ~]# yum install atomic-openshift-utils -y
[root@workstation ~]# cd /usr/share/ansible/openshift-ansible/
[root@workstation openshift-ansible]# ls
inventory playbooks roles

[root@workstation ws]# ansible-playbook
/usr/share/ansible/openshift-ansible/playbooks/prerequisites.yml

[root@workstation ws]# ansible-playbook
/usr/share/ansible/openshift-ansible/playbooks/deploy_cluster.yml