Describing and Exploring OpenShift Networking Concepts

Goal	Describe and explore OpenShift networking concepts.
Objectives	● Describe how OpenShift implements software-defined networking. ● Describe how OpenShift routing works and create a route.

Describing OpenShift's Implementation of Software-Defined Networking

Software-Defined Networking (SDN)

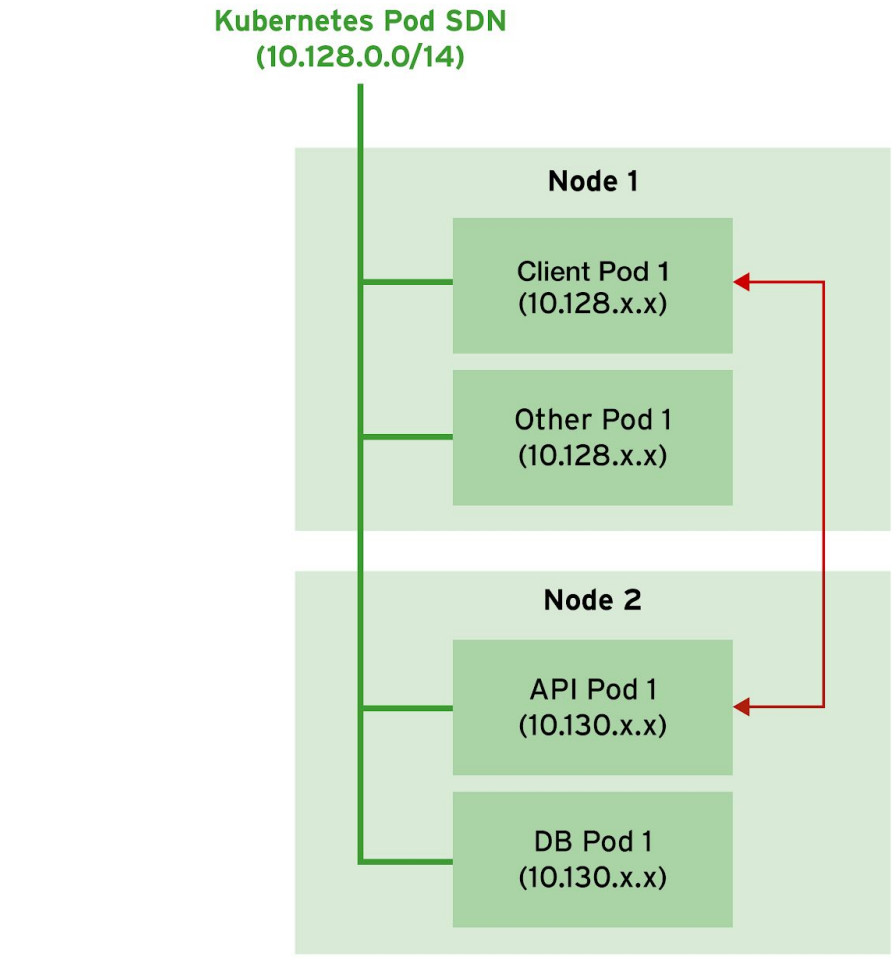
By default, Docker networking uses a host-only virtual bridge, and all containers within a host are attached to it. All containers attached to this bridge can communicate between themselves, but cannot communicate with containers on a different host. Traditionally, this communication is handled using *port mapping* , where container ports are bound to ports on the host and all communication is routed via the ports on the physical host. Manually managing all of the port bindings when you have a large number of hosts with containers is cumbersome and difficult.

To enable communication between containers across the cluster, OpenShift Container Platform uses a *Software-Defined Networking* (*SDN*) approach. Software-Defined networking is a networking model that allows network administrators to manage network services through the abstraction of several networking layers. SDN decouples the software that handles the traffic, called the *control plane* , and the underlying mechanisms that route the traffic, called the *data plane* . SDN enables communication between the control plane and the data plane.

In OpenShift Container Platform 3.9, administrators can configure three SDN plug-ins for the pod network:

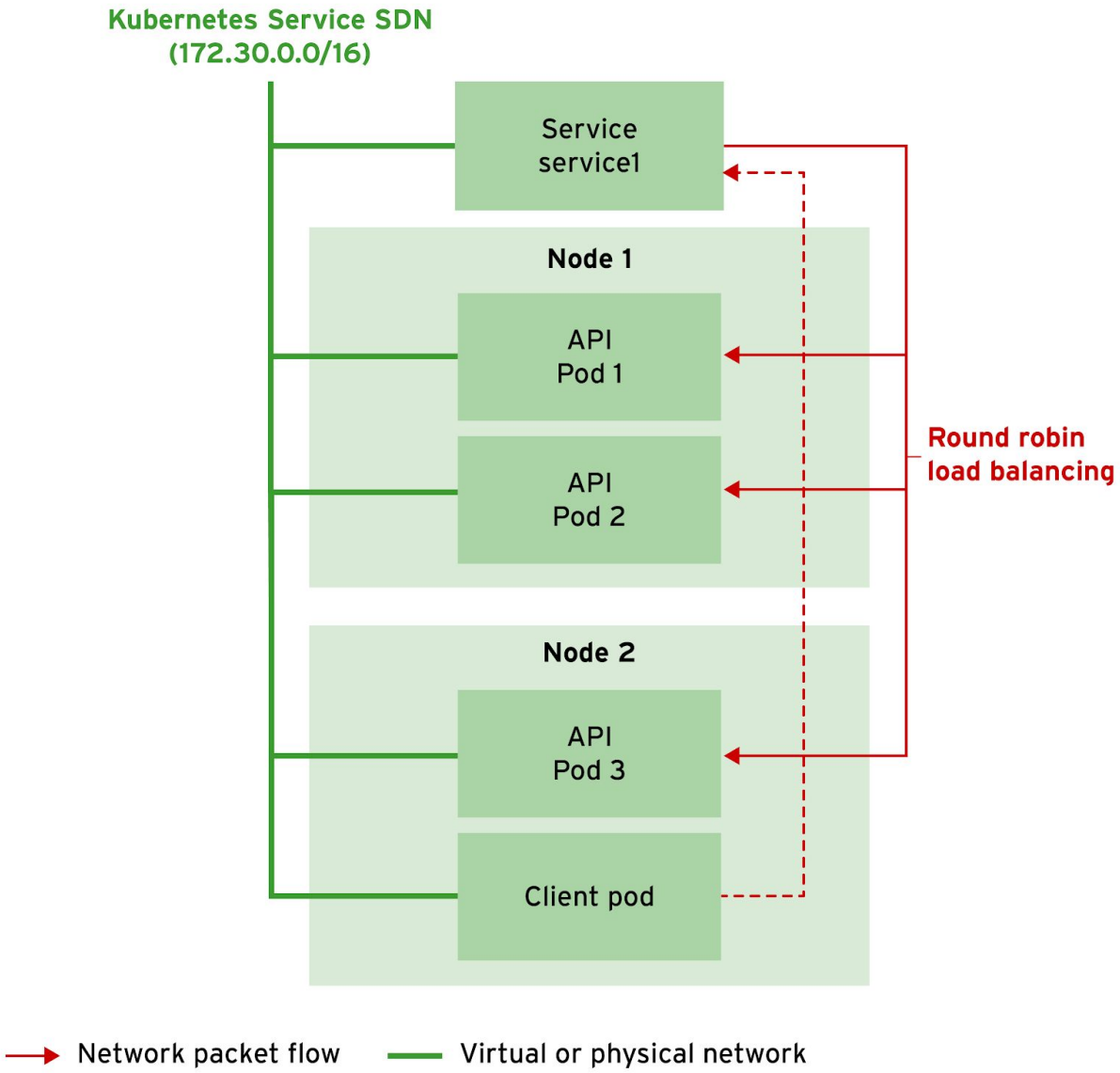
- The`ovs-subnet`plug-in, which is the default plug-in.`ovs-subnet`provides a *flat* pod network where every pod can communicate with every other pod and service.
- The`ovs-multitenant`plug-in provides an extra layer of isolation for pods and services. When using this plug-in, each project receives a unique Virtual Network ID (`VNID`) that identifies traffic from the pods that belong to the project. By using the VNID, pods from different projects cannot communicate with pod and services from a different project.
- The`ovs-networkpolicy`is a plug-in that allows administrators to define their own isolation policies by using the`NetworkPolicy`objects.

The cluster network is established and maintained by OpenShift SDN, which creates an overlay network using Open vSwitch. Master nodes do not have access to containers via the cluster network unless administrators configure them to act as nodes.



In a default OpenShift Container Platform installation, each pod gets a unique IP address. All the containers within a pod behave as if they are on the same host. Giving each pod its own IP address means that pods are treated like physical hosts or virtual machines in terms of port allocation, networking, DNS, load balancing, application configuration, and migration.

Kubernetes provides the concept of a *service* , which is an essential resource in any OpenShift application. A service acts as a load balancer in front of one or more pods. The service provides a stable IP address, and it allows communication with pods without having to keep track of individual pod IP addresses.



Most real-world applications do not run as a single pod. They need to scale horizontally, so an application could run on many pods to meet growing user demand. In an OpenShift cluster, pods are constantly created and destroyed across the nodes in the cluster. Pods get a different IP address each time they are created. Instead of a pod having to discover the IP address of another pod, a service provides a single, unique IP address for other pods to use, independent of where the pods are running. A service load-balances client requests among member pods.

OpenShift Network Topology

The set of pods running behind a service is managed automatically by OpenShift Container Platform. Each service is assigned a unique IP address for clients to connect to. This IP address also comes from the OpenShift SDN and it is distinct from the pod's internal network, but visible only from within the cluster. Each pod matching the `selector` is added to the service resource as an endpoint. As pods are created and killed, the endpoints behind a service are automatically updated.

The following listing shows a minimal service definition in YAML syntax:

```
- apiVersion: v1
  kind: Service
  metadata:
    labels:
      app: hello-openshift
      name: hello-openshift
  spec:
    ports:
    - name: 8080-tcp
      port: 8080
      protocol: TCP
      targetPort: 8080
    selector:       app: hello-openshift
      deploymentconfig: hello-openshift
```

Getting Traffic into and out of the Cluster

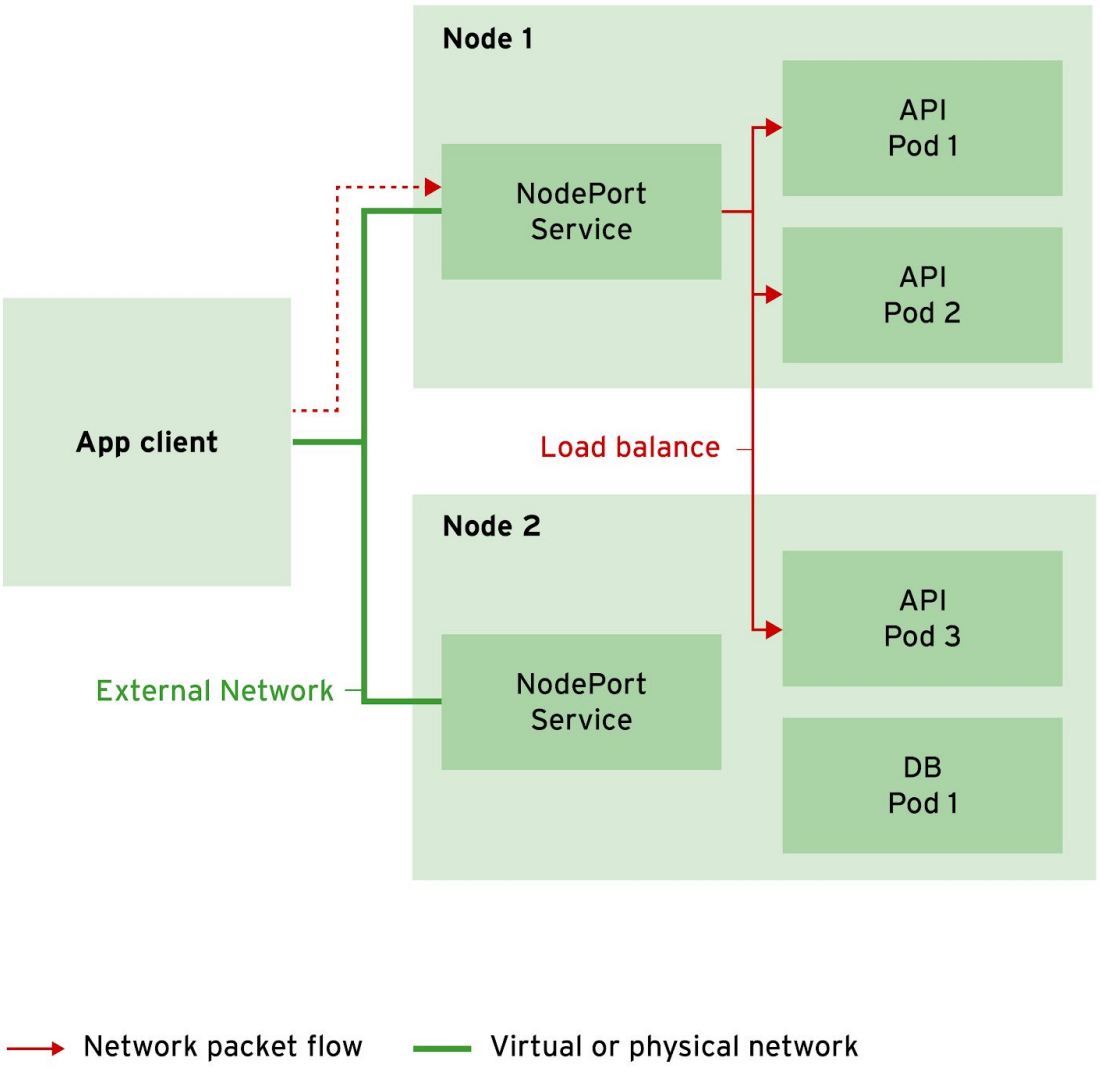
By default, pod and service IP addresses are not reachable from outside the OpenShift cluster. For applications that need access to the service from outside the OpenShift cluster, three methods exist:

- `HostPort/HostNetwork`: In this approach, clients can reach application pods in the cluster directly via the network ports on the host. Ports in the application pod are bound to ports on the host where they are running. This approach requires escalated privileges to run, and there is a risk of port conflicts when there are a large number of pods running in the cluster.
- `NodePort`: This is an older Kubernetes-based approach, where the service is exposed to external clients by binding to available ports on the node host, which then proxies connections to the service IP address. Use the **oc edit svc** command to edit service attributes, specify `NodePort` as the type, and provide a port value for the `nodePort` attribute. OpenShift then proxies connections to the service via the

public IP address of the node host and the port value set in `nodePort`. This approach supports non-HTTP traffic.

- OpenShift `routes`: This is the preferred approach in OpenShift. It exposes services using a unique URL. Use the **oc expose** command to expose a service for external access, or expose a service from the OpenShift web console. In this approach, only HTTP, HTTPS, TLS with SNI, and WebSockets are currently supported.

The following figure shows how `NodePort` services allow external access to Kubernetes services.



The following listing shows a NodePort definition in YAML syntax:

```
apiVersion: v1
kind: Service
metadata:
...
spec:
  ports:
  - name: 3306-tcp
    port: 3306
    protocol: TCP
    targetPort: 3306 nodePort: 30306
  selector:
    app: mysqldb
    deploymentconfig: mysqldb
    sessionAffinity: None
  type: NodePort
...
```

Accessing External Networks

Pods can communicate with external networks using the address of their host. As long as the host can resolve the server that the pod needs to reach, the pods can communicate with the target server using the *network address translation* (NAT) mechanism.

Guided Exercise: Exploring Software-Defined Networking

In this exercise, you will deploy multiple pods of an application and review OpenShift's Software-Defined Networking feature.

Outcomes

You should be able to deploy multiple replicas of an application pod and access them:

- Directly via their pod IP addresses from within the cluster.
- Using the OpenShift service IP address from within the cluster.
- From external clients using node ports from outside the cluster.

Make sure your OpenShift environment is installed and in running state.

1. Create a new project.
From the`workstation`VM, access the OpenShift master at `https://master.lab.example.com`with the OpenShift client.

Log in as `developer` and accept the certificate.

```
[student@workstation ~]$oc login -u developer -p redhat
\https://master.lab.example.com... output omitted ...
      Use insecure connections? (y/n): y
```

2. Create the `network-test-0X` project. (X: Your OpenShift ID)

```
[student@workstation ~]$oc new-project network-test
```

3. Deploy multiple pods of a test application.

Deploy the scaling application from the private registry.

The application runs on port 8080 and displays the IP address of the host. In the environment, this corresponds to the pod IP address running the application:

```
[student@workstation ~]$oc new-app --name=hello -i php:7.0
\http://registry.lab.example.com/scaling
```

Run the following command to verify that the application pod is ready and running. It will take some time to build and deploy the pods.

```
[student@workstation ~]$oc get pods
```

The output from the command should be similar as the following:

```
NAME            READY     STATUS      RESTARTS   AGE
hello-1-build   0/1       Completed   0          30s
      hello-1-nvfgd   1/1       Running     0          23s
```

Run the **oc scale** command to scale the application to two pods.

```
[student@workstation ~]$oc scale --replicas=2 dc hello
      deploymentconfig "hello" scaled
```

You should now see two pods running, typically one pod on each node:

```
[student@workstation ~]$oc get pods -o wide
NAME            ..   STATUS     IP             NODE
hello-1-4bb1t   ..   Running    10.129.0.27    node1.lab.example.com
      hello-1-nvfgd   ..   Running    10.130.0.13    node2.lab.example.com
```

4. Verify that the application is *not* accessible from `workstation`, using the IP addresses listed in the previous step.(ip's may be different in your machine)

```
[student@workstation ~]$curl http://10.129.0.27:8080
curl: (7) Failed connect to 10.129.0.27:8080; Network is unreachable

[student@workstation ~]$curl http://10.130.0.13:8080
curl: (7) Failed connect to 10.130.0.13:8080; Network is unreachable
```

Pod IP addresses are not reachable from outside the cluster.

5. Verify that the application is accessible using the individual pod IP addresses.

Launch two new terminals on `workstation` and connect to `node1` and `node2` using **ssh** .:

```
[student@workstation ~]$ssh root@node1
[student@workstation ~]$ssh root@node2
```

Verify that the application is accessible on `node1` and `node2`, using the respective IP addresses shown in the previous step.

```
[root@node1 ~]#curl http://10.129.0.27:8080
<html>
 <head>
  <title>PHP Test</title>
 </head>
 <body>
 <br/> Server IP: 10.129.0.27
 </body>
</html>
```

```
[root@node2 ~]#curl http://10.130.0.13:8080
<html>
 <head>
  <title>PHP Test</title>
 </head>
 <body>
 <br/> Server IP: 10.130.0.13
 </body>
</html>
```

6. Verify that the application is accessible using the service IP address, which is the cluster IP:

From the `workstation` VM, identify the service IP address using the **oc get svc** command.

```
[student@workstation ~]$oc get svc hello
NAME      CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
      hello     172.30.105.51   <none>        8080/TCP   25m
```

Verify that the application is *not* accessible from `workstation` using the cluster IP address.

```
[student@workstation ~]$curl http://172.30.105.51:8080
      curl: (7) Failed connect to 172.30.105.51:8080; Network is unreachable
```

The cluster IP address is also not reachable from outside the cluster.

Verify that the application is accessible from either `master`, `node1`, or `node2` using the cluster IP address.

```
[student@workstation ~]$ssh root@node1 curl http://172.30.105.51:8080
...
 Server IP: 10.129.0.27
```

```
...
```

Send more HTTP requests to the cluster IP URL, and observe how the requests are load balanced and routed to the two pods in a round-robin manner.

```
[student@workstation ~]$ssh root@node1 curl http://172.30.105.51:8080
...
 <br/> Server IP: 10.130.0.13
...
```

Inspect the service from the application.

Describe the details of the`hello`service using the **oc describe svc** command.

```
[student@workstation ~]$oc describe svc hello
Name:              hello
Namespace:         network-test
Labels:            app=hello
Annotations:       openshift.io/generated-by=OpenShiftNewApp
Selector:          app=hello,deploymentconfig=hello
Type:              ClusterIP
IP:                172.30.171.155
Port:              8080-tcp  8080/TCP
TargetPort:        8080/TCP
Endpoints:         10.128.0.24:8080,10.129.0.20:8080
Session Affinity:  None
Events:            <none>
```

The`Endpoints`attribute displays a list of pod IP addresses that the requests are routed to. These endpoints are automatically updated when pods are killed or when new pods are created.

OpenShift uses the`selectors`and`labels`that are defined for pods to load balance the application with a given cluster IP. OpenShift routes requests for this service to all pods labeled`app=hello`and`deploymentconfig=hello`.

Display the details one of the pods to ensure that labels are present.

```
[student@workstation ~]$oc describe podhello-1-4bb1t
...
Labels: app=hello
        deployment=hello-1
        deploymentconfig=hello
...
```

7. Enable access to the application from outside the cluster.

Edit the service configuration for the application and change the service type to *NodePort* .

Edit the service configuration for the application using the **oc edit svc** command.

```
[student@workstation ~]$oc edit svc hello
```

This command opens a **vi** editor buffer which shows the service configuration in YAML format. Update the`type`of the service to`NodePort`, and add a new attribute

called `nodePort` to the `ports` array with a value of 30800 for the attribute.

```
apiVersion: v1
kind: Service
metadata:
...
spec:
  clusterIP: 172.30.105.51
  ports:
  - name: 8080-tcp
    port: 8080
    protocol: TCP
    targetPort: 8080
    nodePort: 30800
  selector:
    app: hello
    deploymentconfig: hello
  sessionAffinity: None
  type: NodePort
status:
      ...
```

Type **:wq** to save the contents of the buffer and exit the editor.

Verify your changes by running the **oc describe svc** command again.

```
[student@workstation ~]$oc describe svc hello
Name:                   hello
Namespace:              network-test
Labels:                 app=hello
Annotations:            openshift.io/generated-by=OpenShiftNewApp
Selector:               app=hello,deploymentconfig=hello
Type:                   NodePort
IP:                     172.30.171.155
Port:                   8080-tcp  8080/TCP
TargetPort:             8080/TCP
NodePort:               8080-tcp  30800/TCP
Endpoints:              10.128.0.24:8080,10.129.0.20:8080
Session Affinity:       None
External Traffic Policy:  Cluster
Events:                 <none>
```

Access the application using the NodePort IP address from the `workstation` VM.

```
[student@workstation ~]$curl http://node1.lab.example.com:30800
...
  Server IP: 10.129.0.27
...

[student@workstation ~]$curl http://node2.lab.example.com:30800
...
  Server IP: 10.130.0.13
```

```
...
```

The application is accessible from outside the cluster and the requests are still load balanced between the pods.

8. You can see how traffic is routed to pods from external clients. To verify the outgoing traffic, that is, from the pods to the outside world, you can use the **oc rsh** command to open a shell inside the pod as described below.

Use the **oc rsh** command to access the shell inside a pod.

```
[student@workstation ~]$oc rsh hello-1-4bb1t
```

Access machines that are outside the OpenShift cluster from the pod. For example, access the Git repository hosted on the `services` VM to verify that pods can reach it.

```
sh-4.2$curl http://services.lab.example.com
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-US" lang="en-US">
<!-- git web interface version 1.8.3.1, (C) 2005-2006, Kay Sievers
<kay.sievers@vrfy.org>, Christian Gierke -->
<!-- git core binaries version 1.8.3.1 -->
...
```

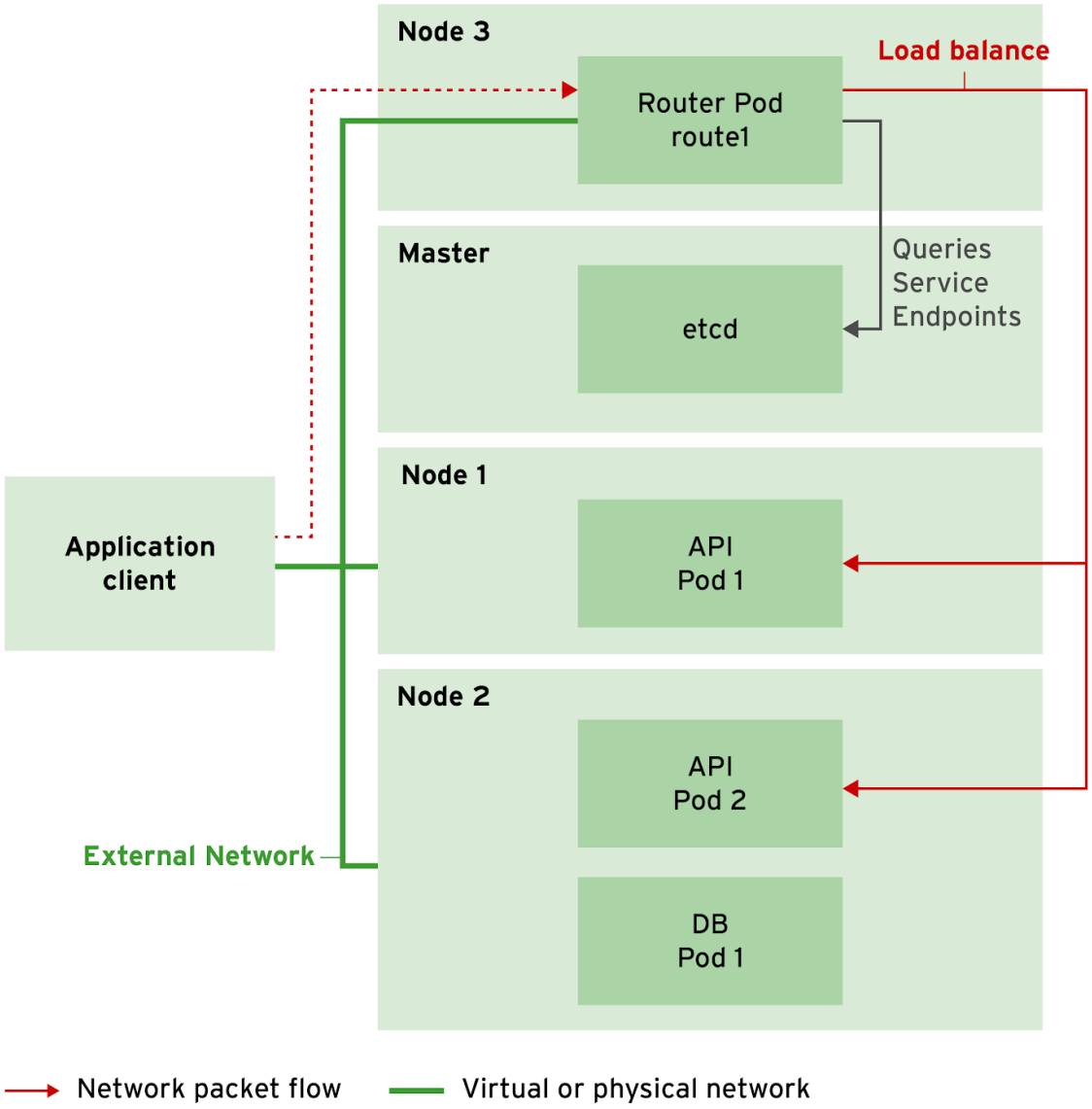
Type **exit** to exit the pod shell and return to the `workstation` prompt.

```
sh-4.2$exit[student@workstation ~]$
```

Creating Routes

Describing the OpenShift Router

While OpenShift services allow for network access between pods inside an OpenShift instance, OpenShift routes allow for network access to pods from outside the OpenShift instance.



A route connects a public-facing IP address and DNS host name to an internal-facing service IP address. At least, this is the concept. In practice, to improve performance and reduce latency, the OpenShift router connects directly to the pods over the networks created by OpenShift Container Platform, using the service only to find the endpoints; that is, the pods exposed by the service.

OpenShift routes are implemented by a shared router service, which runs as a pod inside the OpenShift instance, and can be scaled and replicated like any other regular pod. This router service is based on the open source software *HAProxy* .

An important consideration for OpenShift administrators is that the public DNS host names configured for routes need to point to the public-facing IP addresses of the nodes running the router. Router pods, unlike regular application pods, bind to their nodes' public IP

addresses, instead of to the internal pod network. This is typically configured using a DNS wildcard.

The following listing shows a minimal route defined using YAML syntax:

```
- apiVersion: v1
  kind: Route
  metadata:
    creationTimestamp: null
    labels:
      app: quoteapp
    name: quoteapp
  spec:
    host: quoteapp.apps.lab.example.com
    port:
      targetPort: 8080-tcp
    to:
      kind: Service
      name: quoteapp
```

The kind of Kubernetes resource. In this case, a`Route`.

Resources	
Files:	`/home/student/DO280/labs/secure-route`
Application URL:	`https://hello.apps.lab.example.com`

1. Create a new project.

   From the`workstation`VM, connect to the OpenShift master server accessible at`https://master.lab.example.com`with the OpenShift client.

   Log in as`developer`. If prompted, accept the certificate.

   ```
   [student@workstation ~]$oc login -u developer -p redhat
   \https://master.lab.example.com
   ```

   Create the`secure-route`project.

   ```
   [student@workstation ~]$oc new-project secure-route0X(X: Openhift ID)
   ```

2. Deploy a test application.

Deploy the hello-openshift application from the private registry. The application runs on port 8080 and displays a simple text message.

- `[student@workstation ~]$` **`oc new-app \--docker-image=registry.lab.example.com/openshift/hello-openshift \--name=hello`**

3. Run the following command to verify that the application pod is ready and running. It will take some time to deploy the pods.

```
[student@workstation ~]$oc get pods -o wide
NAME            READY     STATUS    ..  IP              NODE
```

- *`hello-1-qmnbn`* `1/1  Running  ..  10.130.0.11  node1.lab.example.com`

  Make note of the IP address and the node FQDN for the `hello` pod. The name and IP address of the pod on your system might be different. You will need this IP to test the application later in the lab.

Create a self-signed TLS certificate for securing the route.

Briefly review the commands in the `create-cert.sh` file in the `/home/student/DO280/labs/secure-route` directory.

```
[student@workstation ~]$cat \/home/student/DO280/labs/secure-route/create-cert.sh
echo "Generating a private key..."
openssl genrsa -out hello.apps.lab.example.com.key 2048
...

echo "Generating a CSR..."
openssl req -new -key hello.apps.lab.example.com.key \-out
hello.apps.lab.example.com.csr \-subj
"/C=US/ST=NC/L=Raleigh/O=RedHat/OU=RHT/CN=hello.apps.lab.example.com"
...

echo "Generating a certificate..."
openssl x509 -req -days 366 -in \hello.apps.lab.example.com.csr -signkey
\hello.apps.lab.example.com.key \-out hello.apps.lab.example.com.crt
```

- `...`

  The script creates a self-signed TLS certificate that is valid for 366 days.

4. Run the `create-cert.sh` script.

```
[student@workstation ~]$cd
/home/student/DO280/labs/secure-route[student@workstation
secure-route]$./create-cert.sh
Generating a private key…
```

```
Generating RSA private key, 2048 bit long modulus
..............................+++
..............................+++
e is 65537 (0x10001)

Generating a CSR...

Generating a certificate...
Signature ok
subject=/C=US/ST=NC/L=Raleigh/O=RedHat/OU=RHT/CN=hello.apps.lab.example.com
Getting Private key
```

- DONE.

  Verify that three files are created in the same folder:

  - hello.apps.lab.example.com.crt
  - hello.apps.lab.example.com.csr
  - hello.apps.lab.example.com.key

Create a secure edge-terminated route using the generated TLS certificate and key.

Create a new secure edge-terminated route with the files generated in the previous step. From the terminal window, run the following command. The command is available at/home/student/DO280/labs/secure-route/commands.txtfile to minimize typing errors.

```
[student@workstation secure-route]$oc create route edge \--service=hello
--hostname=hello.apps.lab.example.com \--key=hello.apps.lab.example.com.key
\--cert=hello.apps.lab.example.com.crt
      route "hello" created
```

Ensure that the route is created.

```
[student@workstation secure-route]$oc get routes
NAME   HOST/PORT                    SERVICES  PORT      TERMINATION ..
      hello  hello.apps.lab.example.com   hello     8080-tcp  edge ..
```

Inspect the route configuration in YAML format.

```
[student@workstation secure-route]$oc get route/hello -o yaml
apiVersion: v1
kind: Route
metadata:
...
spec:
  host: hello.apps.lab.example.com
  port:
    targetPort: 8080-tcp
  tls:
```

```
    certificate: |
      -----BEGIN CERTIFICATE-----
      MIIDZj...
      -----END CERTIFICATE-----
    key: |
      -----BEGIN RSA PRIVATE KEY-----
      MIIEpQ...
      -----END RSA PRIVATE KEY-----
    termination: edge
  to:
    kind: Service
    name: hello
    weight: 100
  wildcardPolicy: None
status:
      ...
```

Test the route.

Verify that the `hello` service is not accessible using the HTTP URL of the route.

```
[student@workstation secure-route]$curl http://hello.apps.lab.example.com
...
<h1>Application is not available</h1>
      <p>The application is currently not serving requests at this endpoint. It may
not have been started or is still starting.</p>
      ...
```

The generic router home page is displayed, which indicates that the request has not been forwarded to any of the pods.

Verify that the `hello` service is accessible using the secure URL of the route.

```
[student@workstation secure-route]$curl -k -vvv
\https://hello.apps.lab.example.com
* About to connect() to hello.apps.lab.example.com port 443 (#0)
*   Trying 172.25.250.11...
* Connected to hello.apps.lab.example.com (172.25.250.11) port 443 (#0)
...
* SSL connection using TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
* Server certificate:
*      subject: CN=hello.apps.lab.example.com,OU=RHT,O=RedHat,L=Raleigh,ST=NC,C=US
*      start date: Jun 29 07:02:24 2017 GMT
*      expire date: Jun 30 07:02:24 2018 GMT
*      common name: hello.apps.lab.example.com
*      issuer: CN=hello.apps.lab.example.com,OU=RHT,O=RedHat,L=Raleigh,ST=NC,C=US
...
Hello OpenShift!
      ...
```

Because the encrypted traffic is terminated at the router, and the request is forwarded to the pods using unsecured HTTP, you can access the application over plain HTTP using the pod IP address. To do so, use the IP address you noted from the **oc get pods -o wide** command.

Open a new terminal on the`workstation`VM and run the following command.

```
[student@workstation secure-route]$ssh node1 curl -vvv http://10.130.0.11:8080
* About to connect() to 10.130.0.11 port 8080 (#0)
*   Trying 10.130.0.11...
* Connected to 10.130.0.11 (10.130.0.11) port 8080 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.29.0
> Host: 10.130.0.11:8080
> Accept: */*
>
...
Hello OpenShift!
      ...
```

Executing Commands

Overview	
Goal	Execute commands using the command-line interface.
Objectives	● Configure OpenShift resources using the command-line interface. ● Execute commands that assist in troubleshooting common problems.

Configuring Resources with the CLI

Installing the **oc** Command-line Tool

During the OpenShift installation process, the **oc** command-line tool is installed on all master and node machines. You can also install the **oc** client on systems that are not part of

the OpenShift cluster, such as developer machines. When it is installed, you can issue commands after authenticating against any master node with a user name and password

There are several different methods available for installing the **oc** command-line tool, depending on which platform is used:

- On Red Hat Enterprise Linux (RHEL) systems with valid subscriptions, the tool is available as an RPM file and installable using the **yum install** command.
  `[user@host ~]$sudo yum install atomic-openshift-clients`
- For other Linux distributions and other operating systems, such as Windows and macOS, native clients are available for download from the Red Hat Customer Portal. This also requires an active OpenShift subscription. These downloads are statically compiled to reduce incompatibility issues.

Useful Commands to Manage OpenShift Resources

After the **oc** CLI tool has been installed, you can use the **oc help** command to display help information. There are **oc** subcommands for tasks such as:

- Logging in to and out of an OpenShift cluster.
- Creating, changing, and deleting projects.
- Creating applications inside a project.
- Creating a deployment configuration or a build configuration from a container image, and all associated resources.
- Creating, deleting, inspecting, editing, and exporting individual resources, such as pods, services, and routes inside a project.
- Scaling applications.
- Starting new deployments and builds.
- Checking logs from application pods, deployments, and build operations.

You can use the **oc login** command to log in interactively, which prompts you for a server name, a user name, and a password, or you can include the required information on the command line.

```
[student@workstation ~]$oc login https://master.lab.example.com \
-u developer -p redhat
```

After successful authentication from a client, OpenShift saves an authorization token in the user's home folder. This token is used for subsequent requests, negating the need to reenter credentials or the full master URL.

To check your current credentials, run the **oc whoami** command:

```
[student@workstation ~]$oc whoami
```

This command displays the user name that you used when logging in.

```
developer
```

To create a new project, use the **oc new-project** command:

```
[student@workstation ~]$oc new-project working
```

Use run the **oc status** command to verify the status of the project:

```
[student@workstation ~]$oc status
```

Initially, the output from the status command reads:

```
In project working on server https://master.lab.example.com

You have no services, deployment configs, or build configs.
Run 'oc new-app' to create an application.
```

The output of the above command changes as you create new projects and add resources to those projects.

To delete a project, use the **oc delete project** command:

```
[student@workstation ~]$oc delete project working
```

To log out of the OpenShift cluster, use the **oc logout** command:

```
[student@workstation ~]$oc logout
Logged "developer" out on "https://master.lab.example.com"
```

It is possible to log in as the OpenShift cluster administrator from any master node without a password by connecting via **ssh** to the master node.

```
[root@master ~]#oc whoami
system:admin
```

This gives you full privileges over all operations and resources in the OpenShift instance, and should be used with care.

Typically, as an administrator, the **oc get** command is likely the tool that is used most frequently. This allows users to get information about resources in the cluster. Generally, this command displays only the most important characteristics of the resources and omits more detailed information.

If the`RESOURCE_NAME`parameter is omitted, then all resources of the specified`RESOURCE_TYPE`are summarized. The following output is a sample execution of **oc get pods** :

```
NAME                            READY    STATUS       RESTARTS   AGE
docker-registry-1-5r583         1/1      Running      0          1h
trainingrouter-1-l44m7          1/1      Running      0          1h
```

oc get all

If the administrator wants a summary of all of the most important components of the cluster, the **oc get all** command can be executed. This command iterates through the major resource types and prints out a summary of their information. For example:

```
NAME                     DOCKER REPO                     TAGS   UPDATED
is/registry-console      172.30.211.204:5000             3.3    2 days ago

NAME                   REVISION   DESIRED  CURRENT  TRIGGERED BY
dc/docker-registry     4          1        1        config
dc/docker-console      1          1        1        config
dc/router              4          1        1        config

NAME                    DESIRED        CURRENT  READY    AGE
rc/docker-registry      -1     0       0        0        2d
rc/docker-registry      -2     0       0        0        2d
rc/docker-registry      -3     0       0        0        2d
rc/docker-registry      -4     1       1        1        2d
rc/docker-console       -1     1       1        1        2d
rc/docker-router        -1     0       0        0        2d

NAME                       HOST/PORT                                    PATH
SERVICES            PORT              TERMINATION
routes/docker-registry   docker-registry-default.apps.lab.example.com
docker-registry     5000-tcp          passthrough
routes/registry-console  registry-console-default.apps.lab.example.com
registry-console    registry-console  passthrough

NAME                   CLUSTER_IP         EXTERNAL_IP     PORT(S)                   AGE
svc/docker-registry    172.30.211.204     <none>          5000/TCP                  2d
svc/kubernetes         172.30.0.1         <none>          443/TCP,53/UDP,53/TCP     2d
svc/registry-console   172.30.190.103     <none>          9000/TCP                  2d
svc/router             172.230.63.165     <none>          80/TCP,443/TCP,1936/TCP   2d

NAME                             READY     STATUS     RESTARTS    AGE
po/docker-registry-4-ku34r       1/1       Running    3           2d
po/registry-console-1-zxreg      1/1       Running    3           2d
```

```
po/router-1-yhunh              1/1       Running     5           2d
```

A useful option that you can pass to the **oc get** command is the `-w` option, which watches the resultant output in real-time. This is useful, for example, for monitoring the output of an **oc get pods** command continuously instead of running it multiple times from the shell.

oc describe RESOURCE RESOURCE_NAME

If the summaries provided by **oc get** are insufficient, additional information about the resource can be retrieved by using the **oc describe** command. Unlike the **oc get** command, there is no way to iterate through all of the different resources by type. Although most major resources can be described, this functionality is not available across all resources. The following is an example output from describing a pod resource:

```
Name:               docker-registry-4-ku34r
Namespace:          default
Security Policy:    restricted
Node:               node.lab.example.com/172.25.250.11
Start Time:         Mon, 23 Jan 2017 12:17:28 -0500
Labels:             deployment=docker-registry-4
                    deploymentconfig=docker-registry
                    docker-registry=default
Status:             Running
...Output omitted...
No events
```

oc export

Use the **oc export** command to export a definition of a resource. Typical use cases include creating a backup, or to aid in modifying a definition. By default, the **export** command prints out the object representation in YAML format, but this can be changed by providing a `-o` option.

oc create

Use the **oc create** command to create resources from a resource definition. Typically, this is paired with the **oc export** command for editing definitions.

oc delete RESOURCE_TYPE name

Use the **oc delete** command to remove a resource from the OpenShift cluster. Note that a fundamental understanding of the OpenShift architecture is needed here, because deleting managed resources such as pods results in newer instances of those resources being automatically recreated.

oc exec

Use the **oc exec** command to execute commands inside a container. You can use this command to run interactive as well as noninteractive batch commands as part of a script.

oc rsh POD

The **oc rsh** *pod* command opens a remote shell session to a container. This is useful for logging in and investigating issues in a running container.

To log in to a container shell remotely and execute commands, run the following command.

```
[student@workstation ~]$oc rsh <pod>
```

OpenShift Resource Types

Applications in OpenShift Container Platform are composed of resources of different types. The supported types are listed below:

Container

A definition of how to run one or more processes inside a portable Linux environment. Containers are started from an image and are usually isolated from other containers on the same machine.

Image

A layered Linux file system that contains application code, dependencies, and any supporting operating system libraries. An image is identified by a name that can be local to the current cluster, or point to a remote Docker registry (a storage server for images).

Pod

A set of one or more containers that are deployed onto a node and share a unique IP address and volumes (persistent storage). Pods also define the security and runtime policy for each container.

Label

Labels are key-value pairs that can be assigned to any resource in the system for grouping and selection. Many resources use labels to identify sets of other resources.

Volume

Containers are not persistent by default; their contents are cleared when they are restarted. Volumes are mounted file systems available to pods and their containers, and which may be backed by a number of host-local or network-attached storage endpoints. The simplest volume type is`EmptyDir`, which is a temporary directory on a single machine. As an administrator, you can also allow you to request a *Persistent Volume* that is automatically attached to your pods.

Node

Nodes are host systems set up in the cluster to run containers. Nodes are usually managed by administrators and not by end users.

Service

A service is a logical name representing a set of pods. The service is assigned an IP address and a DNS name, and can be exposed externally to the cluster via a port or a route. An environment variable with the name`*SERVICE*_HOST`is automatically injected into other pods.

Route

A route is a DNS entry that is created to point to a service so that it can be accessed from outside the cluster. Administrators can configure one or more routers to handle those routes, typically through a HAProxy load balancer.

Replication Controller

A replication controller maintains a specific number of pods based on a template that matches a set of labels. If pods are deleted (because the node they run on is taken out of service), the controller creates a new copy of that pod. A replication controller is most commonly used to represent a single deployment of part of an application based on a built image.

Deployment Configuration

A deployment configuration defines the template for a pod and manages deploying new images or configuration changes whenever the attributes are changed. A single deployment configuration is usually analogous to a single microservice. Deployment configurations can support many different deployment patterns, including full restart, customizable rolling updates, as well as pre and post life-cycle hooks. Each deployment is represented as a replication controller.

Build Configuration

A build configuration contains a description of how to build source code and a base image into a new image. Builds can be source-based, using builder images for common languages such as Java, PHP, Ruby, or Python, or Docker-based, which create builds from a Dockerfile. Each build configuration has webhooks and can be triggered automatically by changes to their base images.

Build

Builds create new images from source code, other images, Dockerfiles, or binary input. A build is run inside of a container and has the same restrictions that normal pods have. A build usually results in an image being pushed to a Docker registry, but you can also choose to run a post-build test that does not push an image.

Image Streams and Image Stream Tags

An image stream groups sets of related images using tag names. It is analogous to a branch in a source code repository. Each image stream can have one or more tags (the default tag is called "latest") and those tags might point to external Docker registries, to other tags in the same stream, or be controlled to directly point to known images. In addition, images can be pushed to an image stream tag directly via the integrated Docker registry.

Secret

The secret resource can hold text or binary secrets for delivery into your pods. By default, every container is given a single secret which contains a token for accessing the API (with limited privileges) at`/var/run/secrets/kubernetes.io/serviceaccount`. You can create new secrets and mount them in your own pods, as well as reference secrets from builds (for connecting to remote servers), or use them to import remote images into an image stream.

Project

All of the above resources (except nodes) exist inside of a project. Projects have a list of members and their roles, such as`view`,`edit`, or`admin`, as well as a set of security controls on the running pods, and limits on how many resources the project can use. Resource names are unique within a project. Developers may request that projects be created, but administrators control the resources allocated to projects.

Use the **oc types** command for a quick refresher on the concepts and types available.

Creating Applications Using **oc new-app**

Simple applications, complex multitier applications, and microservice applications can be described by a single resource definition file. This file contains many pod definitions, service definitions to connect the pods, replication controllers or deployment configurations to horizontally scale the application pods, persistent volume claims to persist application data, and anything else needed that can be managed by OpenShift.

The **oc new-app** command can be used, with the `-o json` or `-o yaml` option, to create a skeleton resource definition file in JSON or YAML format, respectively. This file can be customized and used to create an application using the **oc create -f <filename>** command, or merged with other resource definition files to create a composite application.

The **oc new-app** command can create application pods to run on OpenShift in many different ways. It can create pods from existing docker images, from Dockerfiles, or from raw source code using the Source-to-Image (S2I) process.

Run the **oc new-app -h** command to understand all the different options available for creating new applications on OpenShift. The most common options are listed below:

Run the following command to create an application. OpenShift pulls the image based on the registries defined by the `ADD_REGISTRY` option of the Docker configuration file.

```
$ oc new-app mysql MYSQL_USER=user MYSQL_PASSWORD=pass MYSQL_DATABASE=testdb -l
db=mysql
```

To create an application based on an image from a private registry:

```
$ oc new-app --docker-image=myregistry.com/mycompany/myapp --name=myapp
```

To create an application based on source code stored in a Git repository:

```
$ oc new-app https://github.com/openshift/ruby-hello-world --name=ruby-hello
```

To create an application based on source code stored in a Git repository and referring to an image stream:

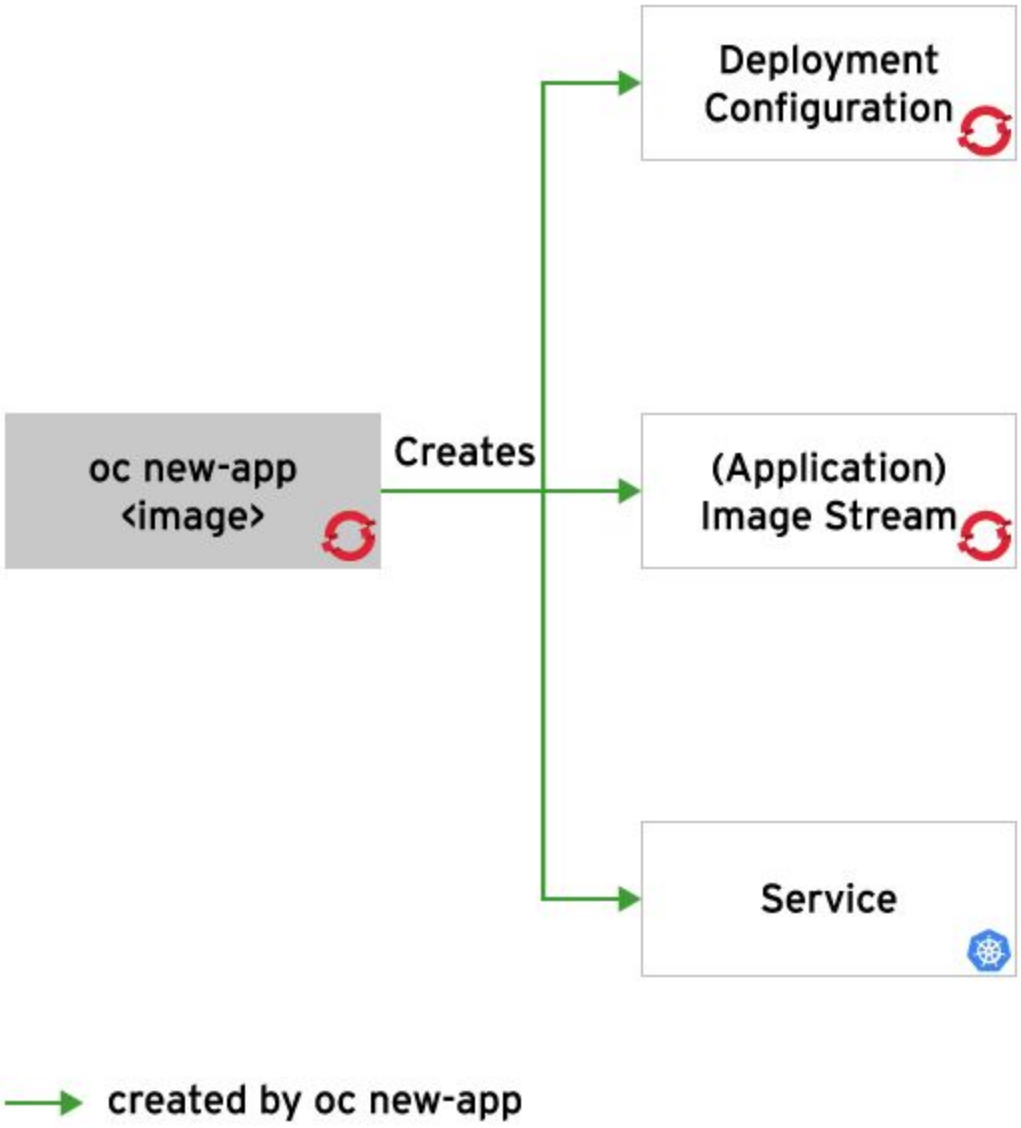
```
$ oc new-app https://mygitrepo/php-hello -i php:7.0 --name=php-hello
```

The **oc new-app** command can create application pods to run on Red Hat OpenShift Container Platform in many different ways. The command can create pods from existing docker images, from Dockerfiles, and from raw source code using the Source-to-Image (S2I) process.

For instance, the following command creates an application based on the `mysql` image from one of the available repositories defined by the `ADD_REGISTRY` directive of the Docker configuration file. The `-l db=mysql` option defines the `db` label with a value of `mysql`.

```
$ oc new-app mysql MYSQL_USER=user \
MYSQL_PASSWORD=pass \
MYSQL_DATABASE=testdb \
-l db=mysql
```

The following figure shows the Kubernetes and OpenShift resources created by the **oc new-app** command when the argument is a container image. The command creates a deployment configuration, an image stream, and a service, which can be accessed externally via a port or a route.



If a source code is used, OpenShift automatically detects the language used and determines the appropriate builder system. However, you can override the builder system to use with the`--strategy=`*`builder`*option:

```
$oc new-app /home/demo/docker/hello --strategy=docker
```

Guided Exercise: Managing an OpenShift Instance Using **oc**

In this exercise, you will manage an instance of OpenShift Container Platform using the **oc** command.

Resources	
Application URL	`https://master.lab.example.com`

1. Get the current status of the OpenShift cluster.

   Open a new terminal on `workstation` and login to OpenShift as the `admin` user with a password of `redhat`. If prompted, access the security certificate.

   ```
   [student@workstation ~]$ oc login -u admin -p redhat
   \https://master.lab.example.com
   ```

Ensure that you are using the `default` project:

```
[student@workstation ~]$ oc project default
      Already on project "default" on server "https://master.lab.example.com:443".
```

List the nodes that are part of the cluster and their status:

```
[student@workstation ~]$ oc get nodes
```

This command produces a tabulated list of nodes similar to the following. Take note of any nodes that have `Ready` as part of their status descriptions. Applications (pods) are deployed on such nodes.

```
NAME                      STATUS    ROLES     AGE       VERSION
master.lab.example.com    Ready     master    8m        v1.9.1+a0ce1bc657
node1.lab.example.com     Ready     compute   8m        v1.9.1+a0ce1bc657
      node2.lab.example.com    Ready     compute   8m        v1.9.1+a0ce1bc657
```

Display more detailed information about the OpenShift master node using the **oc describe** command:

```
[student@workstation ~]$ oc describe node master.lab.example.com
Name:           master.lab.example.com
Role:
Labels:         beta.kubernetes.io/arch=amd64
                beta.kubernetes.io/os=linux
                kubernetes.io/hostname=master.lab.example.com
                node-role.kubernetes.io/master=true
                openshift-infra=apiserver
                region=master

Taints:         <none>
```

```
... output omitted ...
System Info:
... output omitted ...
 Kernel Version:             3.10.0-862.el7.x86_64
 OS Image:                   Red Hat Enterprise Linux Server 7.5 (Maipo)
 Operating System:           linux
 Architecture:               amd64
 Container Runtime Version:  docker://1.13.1
 Kubelet Version:            v1.9.1+a0ce1bc657
 Kube-Proxy Version:         v1.9.1+a0ce1bc657
ExternalID:              master.lab.example.com
... output omitted ...
Events:
... output omitted ...
Normal      Starting          Starting kubelet.
... output omitted ...
      Normal      NodeReady         Node master.lab.example.com status is now:
      NodeReady
```

The `Events` section shows important life-cycle events that have occurred on the master node since the cluster was started. This information is very useful when troubleshooting issues on the master.

Similarly, examine the description of one of the OpenShift nodes:

```
[student@workstation ~]$oc describe node node1.lab.example.com
Name:               node1.lab.example.com
Roles:              compute
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/hostname=node1.lab.example.com
                    node-role.kubernetes.io/compute=true
                    region=infra
Annotations:        volumes.kubernetes.io/controller-managed-attach-detach=true
Taints:             <none>
CreationTimestamp:  Wed, 25 Jul 2018 14:18:36 -0700
... output omitted ...
       Node node1.lab.example.com status is now: NodeReady
```

Inspect the list of existing pods in the project by using the **oc get pods** command.

```
[student@workstation ~]$oc get pods -o wide
NAME                       READY STATUS   IP              NODE
docker-registry-1-pnt4r    1/1   Running  10.128.0.12     node2.lab.example.com
docker-registry-1-q8hrl    1/1   Running  10.129.0.7      node1.lab.example.com
registry-console-1-ch4gp   1/1   Running  10.128.0.11     node2.lab.example.com
router-1-9dq65             1/1   Running  172.25.250.11 node1.lab.example.com
      router-1-vsnb9             1/1   Running  172.25.250.12 node2.lab.example.com
```

The `NODE` column lists the node on which the pod is running.

Use the **oc describe** command to view detailed information about a pod.

```
[student@workstation ~]$oc describe pod docker-registry-1-pnt4r
Name:           docker-registry-1-pnt4r
Namespace:      default
Node:           node2.lab.example.com/172.25.250.12
Start Time:     Wed, 25 Jul 2018 14:21:13 -0700
Labels:         deployment=docker-registry-1
                deploymentconfig=docker-registry
                docker-registry=default
Annotations:    openshift.io/deployment-config.latest-version=1
                openshift.io/deployment-config.name=docker-registry
                openshift.io/deployment.name=docker-registry-1
                openshift.io/scc=restricted
Status:         Running
... output omitted ...
Events:
  Type     Reason                  Age               From
Message
  ----     ------                  ----              ----
-------
  Normal   Scheduled               9m                default-scheduler
Successfully assigned docker-registry-1-pnt4r to node2.lab.example.com
  ... output omitted ...
      Normal   Created                 5m (x3 over 9m)  kubelet,
      node2.lab.example.com  Created container
```

Pay close attention to the`Events`section. It displays important life-cycle related event information about the pod, and is very useful when troubleshooting issues with pods and nodes.

2. Explore the pods.

One of the most useful commands available to the administrator is the **oc exec** command. This command allows the user to execute remote commands against a pod. Run the **hostname** command on the registry pod.

```
[student@workstation ~]$oc exec docker-registry-1-pnt4rhostname
docker-registry-1-pnt4r
```

Run the **ls** command on one of the router pods.

```
[student@workstation ~]$oc exec router-1-9dq65ls /
bin
boot
dev
etc
exports
home
      ...
```

Arbitrary commands can be executed, provided they are available within the pods where you execute them. This ability can be useful for reading files, contents, and processes from within the container itself. Inspect the `/etc/resolv.conf` file.

```
[student@workstation ~]$oc exec docker-registry-1-pnt4rcat /etc/resolv.conf
nameserver 172.25.250.12
search default.svc.cluster.local svc.cluster.local cluster.local lab.example.com
example.com
      options ndots:5
```

Use the **oc rsh** command to initiate a remote shell connection to the router pod, which is useful for more in-depth troubleshooting sessions. On the `master` node, launch a remote shell in the pod:

```
[student@workstation ~]$oc rsh docker-registry-1-pnt4r
      bash-4.2$
```

Run the same **ls** command that was executed before without the interactive shell:

```
bash-4.2$ls /
bin  config.yml etc  lib   lost+found mnt proc     root sbin sys usr
      boot dev        home lib64 media      opt registry run  srv  tmp var
```

Exit the remote shell:

```
bash-4.2$exit
      exit
```

Explore the project status and cluster events.

3. Use the **oc status** command to get a high-level status of the current project:

```
[student@workstation ~]$oc status -v
In project default on server https://master.lab.example.com:443

https://docker-registry-default.apps.lab.example.com (passthrough)
(svc/docker-registry)
  dc/docker-registry deploys
registry.lab.example.com/openshift3/ose-docker-registry:v3.9.14
    deployment #1 deployed 15 minutes ago - 2 pods

svc/kubernetes - 172.30.0.1 ports 443, 53->8053, 53->8053

https://registry-console-default.apps.lab.example.com (passthrough)
(svc/registry-console)
  dc/registry-console deploys docker.io/openshift3/registry-console:v3.9
    deployment #1 deployed 14 minutes ago - 1 pod

svc/router - 172.30.149.232 ports 80, 443, 1936
  dc/router deploys registry.lab.example.com/openshift3/ose-haproxy-router:v3.9.14
    deployment #1 deployed 16 minutes ago - 2 pods
```

- `View details with 'oc describe <resource>/<name>' or list everything with 'oc get all'.`
  The output on your `master` node may be different from that shown above.
- Use the **oc get events** command to view life-cycle events in the OpenShift cluster:
  `[student@workstation ~]$` **`oc get events`**
  Information is presented in a tabular format, in the order in which the events occurred.

Import and export resources.

Use the **oc get all** command to get a list of resources in the project:

```
[student@workstation ~]$oc get all
NAME                                  REVISION DESIRED   CURRENT   TRIGGERED BY
deploymentconfigs/docker-registry     1        2         2         config
deploymentconfigs/registry-console    1        1         1         config
deploymentconfigs/router              1        2         2         config

NAME                              DOCKER REPO                        TAGS UPDATED
imagestreams/registry-console docker-registry.default.svc:
                                  5000/default/\registry-console  v3.9

... output omitted ...

NAME                              READY     STATUS    RESTARTS   AGE
po/docker-registry-1-pnt4r    1/1       Running   2          16m
po/docker-registry-1-q8hrl    1/1       Running   1          16m
po/registry-console-1-ch4gp   1/1       Running   2          15m
po/router-1-9dq65             1/1       Running   1          16m
po/router-1-vsnb9             1/1       Running   2          16m

NAME                     DESIRED   CURRENT   READY     AGE
rc/docker-registry-1     2         2         2         17m
rc/registry-console-1    1         1         1         15m
rc/router-1              2         2         2         17m
```

- `... output omitted ...`
  The output on your system may be different from that shown above.

The **oc export** command exports existing resources and converts them to configuration files (YAML or JSON) for backups, or for recreating resources elsewhere in the cluster.
Export the `docker-registry-1-pnt4r` pod resource in the default YAML format. Replace the pod name with one of the available registry pods in your cluster.

```
[student@workstation ~]$oc export poddocker-registry-1-pnt4r
apiVersion: v1
```

```
kind: Pod
metadata:
  annotations:
    openshift.io/deployment-config.latest-version: "1"
    openshift.io/deployment-config.name: docker-registry
    openshift.io/deployment.name: docker-registry-1
    openshift.io/scc: restricted
  creationTimestamp: null
  generateName: docker-registry-1-
  labels:
    deployment: docker-registry-1
    deploymentconfig: docker-registry
    docker-registry: default
  ownerReferences:
  - apiVersion: v1
    blockOwnerDeletion: true
    controller: true
    kind: ReplicationController
    name: docker-registry-1
```

- *... output omitted ...*

You can also export multiple resources simultaneously as an OpenShift *template* by passing the`--as-template`option to the **oc export** command.

Export the service and deployment configuration definition as a single OpenShift template by running the following command:

```
[student@workstation ~]$oc export svc,dc docker-registry
--as-template=docker-registry
apiVersion: v1
kind: Template
metadata:
  creationTimestamp: null
  name: docker-registry
objects:
- apiVersion: v1
  kind: Service
  metadata:
    creationTimestamp: null
    labels:
      docker-registry: default
    name: docker-registry
  spec:
    ports:
    - name: 5000-tcp
      port: 5000
      protocol: TCP
```

```
      targetPort: 5000
    selector:
      docker-registry: default
    sessionAffinity: ClientIP
    sessionAffinityConfig:
      clientIP:
        timeoutSeconds: 10800
    type: ClusterIP
  status:
    loadBalancer: {}
... output omitted ...
```

The previous command exports both the service definition and the deployment configuration as a template. The output of this command can be sent as input to the **oc create** command to recreate the resource in a cluster.

Run the **oc export --help** command to get a detailed list of options you can pass to the command.

```
[student@workstation ~]$oc export --help... output omitted ...
Examples:
  # export the services and deployment configurations labeled name=test
  oc export svc,dc -l name=test

  # export all services to a template
  oc export service --as-template=test

  # export to JSON
  oc export service -o json
... output omitted ...
```

Executing Troubleshooting Commands

General Environment Information

If you have installed Red Hat OpenShift Container Platform using the RPM installation method, the master and node components will run as native Red Hat Enterprise Linux services. A starting point for data collection from masters and nodes is to use the standard **sosreport** utility that gathers information about the environment along with docker and OpenShift-related information:

```
[root@master ~]#sosreport -k docker.all=on -k docker.logs=on
```

```
sosreport (version 3.5)

This command will collect diagnostic and configuration information from
this Red Hat Enterprise Linux system and installed applications.
...
output omitted ...
 Running plugins.
Please wait ...
  ...
  Running 60/93: openvswitch...
  Running 61/93: origin...
  ...

Creating compressed archive...

Your sosreport has been generated and saved in:
  /var/tmp/sosreport-master.lab.example.com-20180725145249.tar.xz

The checksum is: a544e79319d08538ecfef07687f77e54

Please send this file to your support representative.
```

The **sosreport** command creates a compressed archive containing all the relevant information and saves it in a compressed archive in the `/var/tmp` directory. You can then send this archive file to Red Hat support.

Another useful diagnostic tool for a cluster administrator is the **oc adm diagnostics** command, which gives you the possibility to run several diagnostic checks on the OpenShift cluster including networking, aggregated logging, the internal registry, master and node service checks and many more. Run the **oc adm diagnostics --help** command to get a detailed list of diagnostics that can be run.

OpenShift Troubleshooting Commands

The **oc** command-line client is the primary tool used by administrators to detect and troubleshoot issues in an OpenShift cluster. It has a number of options that enable you to detect, diagnose, and fix issues with masters and nodes, the services, and the resources managed by the cluster. If you have the required permissions, you can directly edit the configuration for most of the managed resources in the cluster.

oc get events

Events allow OpenShift to record information about life-cycle events in a cluster. They allow developers and administrators to view information about OpenShift components in a unified

way. The **oc get events** command provides information about events in an OpenShift namespace. Examples of events that are captured and reported are listed below:

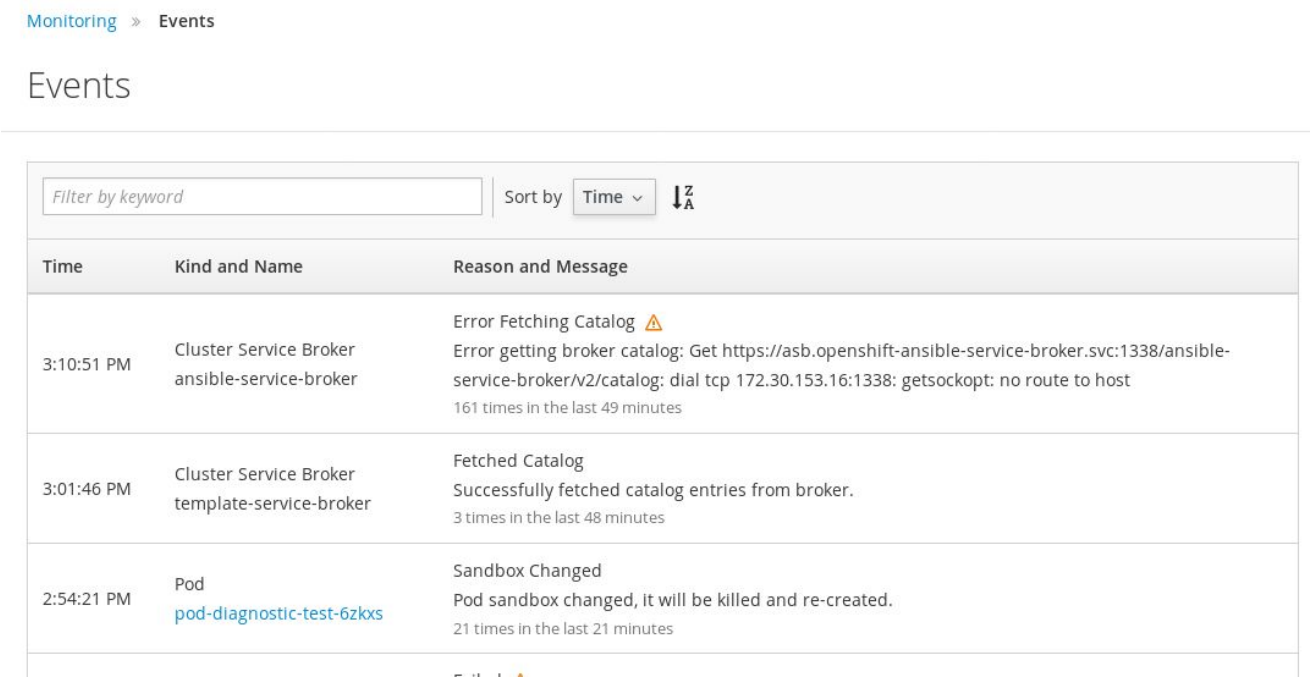
- Pod creation and deletion
- Pod placement scheduling
- Master and node status

Events are useful during troubleshooting. You can get high-level information about failures and issues in the cluster, and then proceed to investigate using log files and other **oc** subcommands.

You can get a list of events in a given project using the following command:

```
[student@workstation ~]$oc get events -n <project>
```

You can also view events in your project from the web console in the Monitoring → Events page. Many other objects, such as pods and deployments, have their own Events tab as well, which shows events related to that object:



oc logs

The **oc logs** command retrieves the log output for a specific build, deployment, or pod. This command works for builds, build configurations, deployment configurations, and pods.

To view the logs for a pod using the **oc** command-line tool:

```
[student@workstation ~]$oc logspod
```

To view the logs for a build:

```
[student@workstation ~]$oc logs bc/build-name
```

Use the **oc logs** command with the `-f` option to follow the log output in real time. This is useful, for example, for monitoring the progress of builds continuously and checking for errors.

You can also view log information about pods, builds, and deployments from the web console.

oc rsync

The **oc rsync** command copies the contents to or from a directory in a running pod. If a pod has multiple containers, you can specify the container ID using the `-c` option. Otherwise, it defaults to the first container in the pod. This is useful for transferring log files and configuration files from the container.

To copy contents from a directory in a pod to a local directory:

```
[student@workstation ~]$oc rsync <pod>:<pod_dir> <local_dir> -c <container>
```

To copy contents from a local directory to a directory in a pod:

```
[student@workstation ~]$oc rsync <local_dir> <pod>:<pod_dir> -c <container>
```

oc port-forward

Use the **oc port-forward** command to forward one or more local ports to a pod. This allows you to listen on a given or random port locally, and have data forwarded to and from given ports in the pod.

The format of this command is as follows:

```
[student@workstation ~]$oc port-forward <pod> [<local_port>:]<remote_port>
```

For example, to listen on port 3306 locally and forward to 3306 in the pod, run the following command:

```
[student@workstation ~]$oc port-forward <pod> 3306:3306
```

Troubleshooting Common Issues

Some of the most common errors and issues seen in OpenShift deployments, and the tools that can be used to troubleshoot them are discussed in the paragraphs below.

Resource Limits and Quota Issues

For projects that have resource limits and quotas set, the improper configuration of resources will cause deployment failures. Use the **oc get events** and **oc describe** commands to investigate the cause of the failure. For example, if you try to create more pods than is allowed in a project with quota restrictions on pod count, you will see the following output when you run the **oc get events** command:

```
14m
Warning  FailedCreate  {hello-1-deploy} Error creating: pods "hello-1" is
forbidden:
exceeded quota: project-quota, requested: cpu=250m, used: cpu=750m, limited:
cpu=900m
```

Source-to-Image (S2I) Build Failures

Use the **oc logs** command to view S2I build failures. For example, to view logs for a build configuration named`hello`:

```
[student@workstation ~]$oc logs bc/hello
```

You can adjust the verbosity of build logs by specifying a`BUILD_LOGLEVEL`environment variable in the build configuration strategy, for example:

```
{
  "sourceStrategy": {
    ...
    "env": [
      {
        "name": "BUILD_LOGLEVEL",
        "value": "5"
      }
    ]
  }
}
```

ErrImagePull and *ImgPullBackOff* Errors

These errors are caused by an incorrect deployment configuration, wrong or missing images being referenced during deployment, or improper Docker configuration. For example:

```
Pod  Warning   FailedSync   {kubelet node1.lab.example.com}
Error syncing pod, skipping: failed to "StartContainer" for "pod-diagnostics" with
ErrImagePull: "image pull failed for
registry.access.redhat.com/openshift3/ose-deployer:v3.5.5.8..."
...
```

```
Pod       spec.containers{pod-diagnostics}   Normal    BackOff {kubelet
node1.lab.example.com}   Back-off pulling image
"registry.access.redhat.com/openshift3/ose-deployer:v3.5.5.8"
...
pod-diagnostic-test-27zqb   Pod  Warning   FailedSync   {kubelet
node1.lab.example.com}
Error syncing pod, skipping: failed to "StartContainer" for "pod-diagnostics" with
ImagePullBackOff: "Back-off pulling image
\"registry.access.redhat.com/openshift3/ose-deployer:v3.5.5.8\""
```

Use the **oc get events** and **oc describe** commands to check for details. Fix deployment configuration errors by editing the deployment configuration using the **oc edit dc/<deploymentconfig>** command.

Incorrect Docker Configuration

Incorrect docker configuration on masters and nodes can cause many errors during deployment. Specifically, check the`ADD_REGISTRY`,`INSECURE_REGISTRY`, and`BLOCK_REGISTRY`settings and ensure that they are valid. Use the **systemctl status** , **oc logs** , **oc get events** , and **oc describe** commands to troubleshoot the issue.

You can change the docker service log levels by adding the`--log-level`parameter for the`OPTIONS`variable in the docker configuration file located at`/etc/sysconfig/docker`. For example, to set the log level to debug:

```
OPTIONS='--insecure-registry=172.30.0.0/16 --selinux-enabled --log-level=debug'
```

Master and Node Service Failures

Run the **systemctl status** command for troubleshooting issues with the`atomic-openshift-master`,`atomic-openshift-node`,`etcd`, and`docker`services. Use the **journalctl -u <unit-name>** command to view the system log for issues related to the previously listed services.

You can increase the verbosity of logging from the`atomic-openshift-node`, the`atomic-openshift-master-controllers`, and`atomic-openshift-master-api`services by editing the`--loglevel`variable in the respective configuration files, and then restarting the associated service.

For example, to set the OpenShift master controller log level to debug, edit the following line in the`/etc/sysconfig/atomic-openshift-master-controllers`file:

```
OPTIONS=--loglevel=4 --listen=https://0.0.0.0:8444
```

Failures in Scheduling Pods

The OpenShift master schedules pods to run on nodes. Sometimes, pods cannot run due to issues with the nodes themselves not being in a *Ready* state, and also due to resource limits and quotas. Use the **oc get nodes** command to verify the status of nodes. During scheduling failures, pods will be in the *Pending* state, and you can check this using the **oc get pods -o wide** command, which also shows the node on which the pod was scheduled to run. Check details about the scheduling failure using the **oc get events** and **oc describe pod** commands.

A sample pod scheduling failure due to insufficient CPU is shown below, as output from the **oc describe** command:

```
{default-scheduler }  Warning  FailedScheduling pod (FIXEDhello-phb4j) failed to
fit in any node
fit failure on node (hello-wx0s): Insufficient cpu
fit failure on node (hello-tgfm): Insufficient cpu
fit failure on node (hello-qwds): Insufficient cpu
```

A sample pod scheduling failure due to a node not being in the *Ready* state is shown below, as output from the **oc describe** command:

```
{default-scheduler }  Warning  FailedScheduling pod (hello-phb4j): no nodes
available to schedule pods
```

Guided Exercise: Troubleshooting Common Problems

Resources	
S2I Application:	`http://services.lab.example.com/php-hello world`

1. Create a new project.

   On the `workstation` host, access the OpenShift master located at `https://master.lab.example.com` with the OpenShift client. Log in as `developer` and accept the security certificate.

   ```
   [student@workstation ~]$oc login -u developer -p redhat
   \https://master.lab.example.com
   ```

Create the `common-troubleshoot` project:

```
[student@workstation ~]$oc new-project common-troubleshoot
```

```
Now using project "common-troubleshoot" on server
"https://master.lab.example.com:443".
```

2. Deploy a Source-to-Image (S2I) application.

Create a new application in OpenShift using the source code from the php-helloworld application, available in the Git repository running on the `services` VM.

```
[student@workstation ~]$oc new-app --name=hello -i php:5.4
\http://services.lab.example.com/php-helloworld
error: multiple images or templates matched "php:5.4": 2

The argument "php:5.4" could apply to the following Docker images, OpenShift image
streams, or templates:

* Image stream "php" (tag "7.0") in project "openshift"
  Use --image-stream="openshift/php:7.0" to specify this image or template

* Image stream "php" (tag "5.6") in project "openshift"
    Use --image-stream="openshift/php:5.6" to specify this image or template
```

Observe the error that informs you about the wrong image stream tag.

List the valid tags in the `php` image stream using the **oc describe** command.

```
[student@workstation ~]$oc describe is php -n openshift
Name:     php
Namespace:    openshift
Created:    About an hour ago
Labels:     <none>
Annotations:    openshift.io/display-name=PHP
      openshift.io/image.dockerRepositoryCheck=2018-07-25T21:16:14Z
Docker Pull Spec: docker-registry.default.svc:5000/openshift/php
Image Lookup:   local=false
Unique Images:    2
Tags:     5

7.1 (latest)
  tagged from registry.lab.example.com/rhscl/php-71-rhel7:latest

  Build and run PHP 7.1 applications on RHEL 7. For more information about using
this builder image, including OpenShift considerations, see
https://github.com/sclorg/s2i-php-container/blob/master/7.1/README.md.
  Tags: builder, php
  Supports: php:7.1, php
  Example Repo: https://github.com/openshift/cakephp-ex.git

  ! error: Import failed (NotFound): dockerimage.image.openshift.io
"registry.lab.example.com/rhscl/php-71-rhel7:latest" not found

7.0
```

```
  tagged from registry.lab.example.com/rhscl/php-70-rhel7:latest

  Build and run PHP 7.0 applications on RHEL 7. For more information about using
this builder image, including OpenShift considerations, see
https://github.com/sclorg/s2i-php-container/blob/master/7.0/README.md.
  Tags: builder, php
  Supports: php:7.0, php
  Example Repo: https://github.com/openshift/cakephp-ex.git

  *
registry.lab.example.com/rhscl/php-70-rhel7@sha256:23765e00df8d0a934ce4f2e22802bc02
11a6d450bfbb69144b18cb0b51008cdd
      5 days ago

5.6
  tagged from registry.lab.example.com/rhscl/php-56-rhel7:latest

  Build and run PHP 5.6 applications on RHEL 7. For more information about using
this builder image, including OpenShift considerations, see
https://github.com/sclorg/s2i-php-container/blob/master/5.6/README.md.
  Tags: builder, php
  Supports: php:5.6, php
  Example Repo: https://github.com/openshift/cakephp-ex.git

  *
registry.lab.example.com/rhscl/php-56-rhel7@sha256:920c2cf85b5da5d0701898f0ec9ee567
473fa4b9af6f3ac5b2b3f863796bbd68


5.5
  tagged from registry.lab.example.com/openshift3/php-55-rhel7:latest

  Build and run PHP 5.5 applications on RHEL 7. For more information about using
this builder image, including OpenShift considerations, see
https://github.com/sclorg/s2i-php-container/blob/master/5.5/README.md.
  Tags: hidden, builder, php
  Supports: php:5.5, php
  Example Repo: https://github.com/openshift/cakephp-ex.git

  ! error: Import failed (NotFound): dockerimage.image.openshift.io
"registry.lab.example.com/openshift3/php-55-rhel7:latest" not found
      About an hour ago
```

The output of the command shows that `php:7.0` and `php-5.6` are a valid tag, whereas `php-7.1` and `php-5.5` are invalid, because those images are not available.

Deploy the application with the correct image stream tag:

```
[student@workstation ~]$oc new-app --name=hello -i php:7.0
\http://services.lab.example.com/php-helloworld
--> Found image c101534 (10 months old) in image stream "openshift/php" under tag
"7.0" for "php:7.0"
... output omitted ...
--> Success
    Build scheduled, use 'oc logs -f bc/hello' to track its progress.
    Application is not exposed. You can expose services to the outside world by
executing one or more of the commands below:
     'oc expose svc/hello'
         Run 'oc status' to view your app.
```

The **oc new-app** command should now succeed.

Verify that the application successfully built and deployed:

```
[student@workstation ~]$oc get pods -o wide
NAME            READY     STATUS    RESTARTS   AGE       IP         NODE
      hello-1-build   0/1       Pending   0          41s       <none>    <none>
```

The `hello-1-build` pod is in the `Pending` state and the application pod is not starting. Investigate why the deployment is in the `Pending` state and fix the issue.

3. Check the logs of the builder pod by using the **oc logs** command.

```
[student@workstation ~]$oc logs hello-1-build
```

This command does not produce any output. The logs show no useful information that can help you troubleshoot the issue.

Check the event log for the project. You can do this in two ways. One way is to use the **oc get events** command as follows:

```
[student@workstation ~]$oc get events
LAST SEEN   FIRST SEEN   COUNT     NAME                                KIND
5m          5m           6         hello-1-build.1544bd6f20c095e9   Pod
5m          5m           1         hello-1-build.1544bd76256c9383   Pod
9s          2m           14        hello-1-build.1544bd9af1aac506   Pod

TYPE      REASON              SOURCE
Warning   FailedScheduling    default-scheduler
Warning   FailedScheduling    default-scheduler
Warning   FailedScheduling    default-scheduler

MESSAGE
0/3 nodes are available: 1 MatchNodeSelector, 2 NodeNotReady.
skip schedule deleting pod: common-troubleshoot/hello-1-build
0/3 nodes are available: 1 MatchNodeSelector, 2 NodeNotReady.
```

Use the **oc describe** command to see if the output gives some hints on why the pod is failing:

```
[student@workstation ~]$oc describe pod hello-1-build
Name:           hello-1-build
Namespace:      common-troubleshoot
Node:           <none>
Labels:         openshift.io/build.name=hello-1
Annotations:    openshift.io/build.name=hello-1
                openshift.io/scc=privileged
Status:         Pending
... output omitted ...
Events:
  Type     Reason            Age                From               Message
  ----     ------            ----               ----               -------
  Warning  FailedScheduling  2s (x18 over 4m)   default-scheduler  0/3 nodes are
available: 1 MatchNodeSelector, 2 NodeNotReady.
```

This command also reports the same`FailedScheduling`warning in the`Events`section.

The event log shows that no nodes are available for scheduling pods to run.

4. Investigate the cause of this warning. Check the status of the nodes in the cluster to see if there are issues. Note that this command should be run as the`root`user on`master`.

```
[student@workstation ~]$ssh root@master oc get nodes
NAME                     STATUS     ROLES     AGE       VERSION
master.lab.example.com   Ready      master    1h        v1.9.1+a0ce1bc657
node1.lab.example.com    NotReady   compute   1h        v1.9.1+a0ce1bc657
node2.lab.example.com    NotReady   compute   1h        v1.9.1+a0ce1bc657
```

The`STATUS`column indicates that both`node1`and`node2`are in the`NotReady`state. Kubernetes cannot schedule pods to run on nodes that are marked as`NotReady`.

5. Investigate why the nodes are not in the`Ready`state. OpenShift nodes must be running the`atomic-openshift-node`service. This service is responsible for communicating with the master, and runs pods on demand when scheduled by the master.
Open two new terminals on`workstation`and log in to the`node1`and`node2`hosts as`root`using the **ssh** command:

```
[student@workstation ~]$ssh root@node1[student@workstation ~]$ssh root@node2
```

Check the status of the`atomic-openshift-node`service on both nodes:

```
[root@node1 ~]#systemctl status atomic-openshift-node.service -l[root@node2
~]#systemctl status atomic-openshift-node.service -l
```

Although both nodes are reporting that the service is active and running, the service reports that something is wrong with the `docker` daemon on the nodes:

```
... output omitted ...
Jul 25 15:46:08 node2.lab.example.com atomic-openshift-node[23635]: E0725
15:46:08.480373   23635 generic.go:197] GenericPLEG: Unable to retrieve pods: rpc
error: code = Unknown desc = Cannot connect to the Docker daemon at
unix:///var/run/docker.sock. Is the docker daemon running?
... output omitted ...
```

6. Check the status of the `docker` service on both nodes:

```
[root@node1 ~]#systemctl status docker.service -l[root@node2 ~]#systemctl status
docker.service -l
```

The service is inactive on both nodes:

```
... output omitted ...
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; vendor preset:
disabled)
   Active: inactive (dead) since Wed 2018-07-25 15:32:35 PDT; 14min ago
... output omitted ...
Jul 25 15:32:35 node2.lab.example.com systemd[1]: Stopped Docker Application
Container Engine.
```

7. Start the `docker` service on both nodes:

```
[root@node1 ~]#systemctl start docker.service[root@node2 ~]#systemctl start
docker.service
```

8. On the `workstation` host, check that the **oc get nodes** command shows both nodes in the `Ready` state:

```
[student@workstation ~]$ssh root@master oc get nodes
NAME                     STATUS    ROLES     AGE       VERSION
master.lab.example.com   Ready     master    1h        v1.9.1+a0ce1bc657
node1.lab.example.com    Ready     compute   1h        v1.9.1+a0ce1bc657
```

9. From the `workstation` VM, verify that the pod is now in the `Running` state:

```
[student@workstation ~]$oc get pods
NAME            READY     STATUS    RESTARTS   AGE
hello-1-build   1/1       Running   0          11m
```

You should see the application pod in the `Running` state.

Verify that the application built and was pushed to the OpenShift internal registry by running the **oc describe is** command:

```
[student@workstation ~]$oc describe is
Name:     hello
```

```
Namespace:    common-troubleshoot
Created:    13 minutes ago
Labels:     app=hello
Annotations:    openshift.io/generated-by=OpenShiftNewApp
Docker Pull Spec: docker-registry.default.svc:5000/common-troubleshoot/hello
Image Lookup:   local=false
Unique Images:    1
Tags:     1

latest
  no spec tag

  *
docker-registry.default.svc:5000/common-troubleshoot/hello@sha256:1aad0df1a216b6b07
0ea3ecfd8cadfdee6dd10b451b8e252dbc835148fc9faf0
      About a minute ago
```