OpenShift Container Platform 4.11

Specialized hardware and driver enablement

Learn about hardware enablement on OpenShift Container Platform
Learn about hardware enablement on OpenShift Container Platform
Abstract

This document provides an overview of hardware enablement in OpenShift Container Platform.
# Table of Contents

## CHAPTER 1. ABOUT SPECIALIZED HARDWARE AND DRIVER ENABLEMENT ................................................. 4

## CHAPTER 2. DRIVER TOOLKIT ........................................................................................................ 5
  2.1. ABOUT THE DRIVER TOOLKIT
      Background
      Purpose
  2.2. PULLING THE DRIVER TOOLKIT CONTAINER IMAGE
      2.2.1. Pulling the Driver Toolkit container image from registry.redhat.io
      2.2.2. Finding the Driver Toolkit image URL in the payload
  2.3. USING THE DRIVER TOOLKIT
      2.3.1. Build and run the simple-kmod driver container on a cluster
  2.4. ADDITIONAL RESOURCES

## CHAPTER 3. SPECIAL RESOURCE OPERATOR .................................................................................. 12
  3.1. ABOUT THE SPECIAL RESOURCE OPERATOR
  3.2. INSTALLING THE SPECIAL RESOURCE OPERATOR
      3.2.1. Installing the Special Resource Operator by using the CLI
      3.2.2. Installing the Special Resource Operator by using the web console
  3.3. USING THE SPECIAL RESOURCE OPERATOR
      3.3.1. Building and running the simple-kmod SpecialResource by using a config map
      3.3.2. Building and running the simple-kmod SpecialResource for a hub-and-spoke topology
  3.4. PROMETHEUS SPECIAL RESOURCE OPERATOR METRICS
  3.5. ADDITIONAL RESOURCES

## CHAPTER 4. NODE FEATURE DISCOVERY OPERATOR ..................................................................... 29
  4.1. ABOUT THE NODE FEATURE DISCOVERY OPERATOR
  4.2. INSTALLING THE NODE FEATURE DISCOVERY OPERATOR
      4.2.1. Installing the NFD Operator using the CLI
      4.2.2. Installing the NFD Operator using the web console
  4.3. USING THE NODE FEATURE DISCOVERY OPERATOR
      4.3.1. Create a NodeFeatureDiscovery instance using the CLI
      4.3.2. Create a NodeFeatureDiscovery CR using the web console
  4.4. CONFIGURING THE NODE FEATURE DISCOVERY OPERATOR
      4.4.1. core
          core.sleepInterval
          core.sources
          core.labelWhiteList
          core.noPublish
          core.klog
              core.klog.addDirHeader
              core.klog.alsoLogToStderr
              core.klog.logBacktraceAt
              core.klog.logDir
              core.klog.logFile
              core.klog.logFileSize
              core.klog.logToStderr
              core.klog.skipHeaders
              core.klog.skipLogHeaders
              core.klog.stderrThreshold
              core.klog.v
              core.klog.vmodule
      4.4.2. sources

---

### Table of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER 1. ABOUT SPECIALIZED HARDWARE AND DRIVER ENABLEMENT</td>
<td>4</td>
</tr>
<tr>
<td>CHAPTER 2. DRIVER TOOLKIT</td>
<td>5</td>
</tr>
<tr>
<td>2.1. ABOUT THE DRIVER TOOLKIT</td>
<td>5</td>
</tr>
<tr>
<td>2.2. PULLING THE DRIVER TOOLKIT CONTAINER IMAGE</td>
<td>6</td>
</tr>
<tr>
<td>2.2.1. Pulling the Driver Toolkit container image from registry.redhat.io</td>
<td>6</td>
</tr>
<tr>
<td>2.2.2. Finding the Driver Toolkit image URL in the payload</td>
<td>6</td>
</tr>
<tr>
<td>2.3. USING THE DRIVER TOOLKIT</td>
<td>7</td>
</tr>
<tr>
<td>2.3.1. Build and run the simple-kmod driver container on a cluster</td>
<td>7</td>
</tr>
<tr>
<td>2.4. ADDITIONAL RESOURCES</td>
<td>11</td>
</tr>
<tr>
<td>CHAPTER 3. SPECIAL RESOURCE OPERATOR</td>
<td>12</td>
</tr>
<tr>
<td>3.1. ABOUT THE SPECIAL RESOURCE OPERATOR</td>
<td>12</td>
</tr>
<tr>
<td>3.2. INSTALLING THE SPECIAL RESOURCE OPERATOR</td>
<td>12</td>
</tr>
<tr>
<td>3.2.1. Installing the Special Resource Operator by using the CLI</td>
<td>12</td>
</tr>
<tr>
<td>3.2.2. Installing the Special Resource Operator by using the web console</td>
<td>13</td>
</tr>
<tr>
<td>3.3. USING THE SPECIAL RESOURCE OPERATOR</td>
<td>14</td>
</tr>
<tr>
<td>3.3.1. Building and running the simple-kmod SpecialResource by using a config map</td>
<td>14</td>
</tr>
<tr>
<td>3.3.2. Building and running the simple-kmod SpecialResource for a hub-and-spoke topology</td>
<td>20</td>
</tr>
<tr>
<td>3.4. PROMETHEUS SPECIAL RESOURCE OPERATOR METRICS</td>
<td>27</td>
</tr>
<tr>
<td>3.5. ADDITIONAL RESOURCES</td>
<td>28</td>
</tr>
<tr>
<td>CHAPTER 4. NODE FEATURE DISCOVERY OPERATOR</td>
<td>29</td>
</tr>
<tr>
<td>4.1. ABOUT THE NODE FEATURE DISCOVERY OPERATOR</td>
<td>29</td>
</tr>
<tr>
<td>4.2. INSTALLING THE NODE FEATURE DISCOVERY OPERATOR</td>
<td>29</td>
</tr>
<tr>
<td>4.2.1. Installing the NFD Operator using the CLI</td>
<td>29</td>
</tr>
<tr>
<td>4.2.2. Installing the NFD Operator using the web console</td>
<td>30</td>
</tr>
<tr>
<td>4.3. USING THE NODE FEATURE DISCOVERY OPERATOR</td>
<td>31</td>
</tr>
<tr>
<td>4.3.1. Create a NodeFeatureDiscovery instance using the CLI</td>
<td>31</td>
</tr>
<tr>
<td>4.3.2. Create a NodeFeatureDiscovery CR using the web console</td>
<td>34</td>
</tr>
<tr>
<td>4.4. CONFIGURING THE NODE FEATURE DISCOVERY OPERATOR</td>
<td>34</td>
</tr>
<tr>
<td>4.4.1. core</td>
<td>34</td>
</tr>
<tr>
<td>core.sleepInterval</td>
<td>34</td>
</tr>
<tr>
<td>core.sources</td>
<td>34</td>
</tr>
<tr>
<td>core.labelWhiteList</td>
<td>35</td>
</tr>
<tr>
<td>core.noPublish</td>
<td>35</td>
</tr>
<tr>
<td>core.klog</td>
<td>35</td>
</tr>
<tr>
<td>core.klog.addDirHeader</td>
<td>35</td>
</tr>
<tr>
<td>core.klog.alsoLogToStderr</td>
<td>35</td>
</tr>
<tr>
<td>core.klog.logBacktraceAt</td>
<td>35</td>
</tr>
<tr>
<td>core.klog.logDir</td>
<td>36</td>
</tr>
<tr>
<td>core.klog.logFile</td>
<td>36</td>
</tr>
<tr>
<td>core.klog.logFileSize</td>
<td>36</td>
</tr>
<tr>
<td>core.klog.logToStderr</td>
<td>36</td>
</tr>
<tr>
<td>core.klog.skipHeaders</td>
<td>36</td>
</tr>
<tr>
<td>core.klog.skipLogHeaders</td>
<td>36</td>
</tr>
<tr>
<td>core.klog.stderrThreshold</td>
<td>36</td>
</tr>
<tr>
<td>core.klog.v</td>
<td>36</td>
</tr>
<tr>
<td>core.klog.vmodule</td>
<td>37</td>
</tr>
<tr>
<td>4.4.2. sources</td>
<td>37</td>
</tr>
</tbody>
</table>
sources.cpu.cpuid.attributeBlacklist  37
sources.cpu.cpuid.attributeWhitelist  37
sources.kernel.kconfigFile  37
sources.kernel.configOpts  38
sources.pci.deviceClassWhitelist  38
sources.pci.deviceLabelFields  38
sources.usb.deviceClassWhitelist  38
sources.usb.deviceLabelFields  38
sources.custom  39

4.5. USING THE NFD TOPOLOGY UPDATER  39
4.5.1. NodeResourceTopology CR  39
4.5.2. NFD Topology Updater command line flags  40
   -ca-file  40
   -cert-file  41
   -h, -help  41
   -key-file  41
   -kubelet-config-file  41
   -no-publish  41
4.5.2.1. -oneshot  42
   -podresources-socket  42
   -server  42
   -server-name-override  42
   -sleep-interval  42
   -version  43
   -watch-namespace  43
CHAPTER 1. ABOUT SPECIALIZED HARDWARE AND DRIVER ENABLEMENT

Many applications require specialized hardware or software that depends on kernel modules or drivers. You can use driver containers to load out-of-tree kernel modules on Red Hat Enterprise Linux CoreOS (RHCOS) nodes. To deploy out-of-tree drivers during cluster installation, use the `kmods-via-containers` framework. To load drivers or kernel modules on an existing OpenShift Container Platform cluster, OpenShift Container Platform offers several tools:

- The Driver Toolkit is a container image that is a part of every OpenShift Container Platform release. It contains the kernel packages and other common dependencies that are needed to build a driver or kernel module. The Driver Toolkit can be used as a base image for driver container image builds on OpenShift Container Platform.

- The Special Resource Operator (SRO) orchestrates the building and management of driver containers to load kernel modules and drivers on an existing OpenShift or Kubernetes cluster.

- The Node Feature Discovery (NFD) Operator adds node labels for CPU capabilities, kernel version, PCIe device vendor IDs, and more.
CHAPTER 2. DRIVER TOOLKIT

Learn about the Driver Toolkit and how you can use it as a base image for driver containers for enabling special software and hardware devices on Kubernetes.

IMPORTANT
The Driver Toolkit is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

2.1. ABOUT THE DRIVER TOOLKIT

Background
The Driver Toolkit is a container image in the OpenShift Container Platform payload used as a base image on which you can build driver containers. The Driver Toolkit image contains the kernel packages commonly required as dependencies to build or install kernel modules, as well as a few tools needed in driver containers. The version of these packages will match the kernel version running on the Red Hat Enterprise Linux CoreOS (RHCOS) nodes in the corresponding OpenShift Container Platform release.

Driver containers are container images used for building and deploying out-of-tree kernel modules and drivers on container operating systems like RHCOS. Kernel modules and drivers are software libraries running with a high level of privilege in the operating system kernel. They extend the kernel functionalities or provide the hardware-specific code required to control new devices. Examples include hardware devices like Field Programmable Gate Arrays (FPGA) or GPUs, and software-defined storage (SDS) solutions, such as Lustre parallel file systems, which require kernel modules on client machines. Driver containers are the first layer of the software stack used to enable these technologies on Kubernetes.

The list of kernel packages in the Driver Toolkit includes the following and their dependencies:

- **kernel-core**
- **kernel-devel**
- **kernel-headers**
- **kernel-modules**
- **kernel-modules-extra**

In addition, the Driver Toolkit also includes the corresponding real-time kernel packages:

- **kernel-rt-core**
- **kernel-rt-devel**
- **kernel-rt-modules**
- **kernel-rt-modules-extra**
The Driver Toolkit also has several tools which are commonly needed to build and install kernel modules, including:

- elfutils-libelf-devel
- kmod
- binutils-kabi-dw
- kernel-abi-whitelists
- dependencies for the above

**Purpose**

Prior to the Driver Toolkit’s existence, you could install kernel packages in a pod or build config on OpenShift Container Platform using entitled builds or by installing from the kernel RPMs in the hosts machine-os-content. The Driver Toolkit simplifies the process by removing the entitlement step, and avoids the privileged operation of accessing the machine-os-content in a pod. The Driver Toolkit can also be used by partners who have access to pre-released OpenShift Container Platform versions to prebuild driver-containers for their hardware devices for future OpenShift Container Platform releases.

The Driver Toolkit is also used by the Special Resource Operator (SRO), which is currently available as a community Operator on OperatorHub. SRO supports out-of-tree and third-party kernel drivers and the support software for the underlying operating system. Users can create recipes for SRO to build and deploy a driver container, as well as support software like a device plug-in, or metrics. Recipes can include a build config to build a driver container based on the Driver Toolkit, or SRO can deploy a prebuilt driver container.

### 2.2. PULLING THE DRIVER TOOLKIT CONTAINER IMAGE

The driver-toolkit image is available from the Container images section of the Red Hat Ecosystem Catalog and in the OpenShift Container Platform release payload. The image corresponding to the most recent minor release of OpenShift Container Platform will be tagged with the version number in the catalog. The image URL for a specific release can be found using the `oc adm` CLI command.

#### 2.2.1. Pulling the Driver Toolkit container image from registry.redhat.io

Instructions for pulling the driver-toolkit image from registry.redhat.io with podman or in OpenShift Container Platform can be found on the Red Hat Ecosystem Catalog. The driver-toolkit image for the latest minor release will be tagged with the minor release version on registry.redhat.io for example registry.redhat.io/openshift4/driver-toolkit-rhel8:v4.11.

#### 2.2.2. Finding the Driver Toolkit image URL in the payload

**Prerequisites**

- You obtained the image pull secret from the Red Hat OpenShift Cluster Manager.
- You installed the OpenShift CLI (oc).

**Procedure**

1. The image URL of the driver-toolkit corresponding to a certain release can be extracted from the release image using the `oc adm` command:
2.3. USING THE DRIVER TOOLKIT

As an example, the Driver Toolkit can be used as the base image for building a very simple kernel module called simple-kmod.

**NOTE**

The Driver Toolkit contains the necessary dependencies, openssl, mokutil, and keyutils, needed to sign a kernel module. However, in this example, the simple-kmod kernel module is not signed and therefore cannot be loaded on systems with Secure Boot enabled.

2.3.1. Build and run the simple-kmod driver container on a cluster

**Prerequisites**

- You have a running OpenShift Container Platform cluster.
- You set the Image Registry Operator state to Managed for your cluster.
- You installed the OpenShift CLI (`oc`).
- You are logged into the OpenShift CLI as a user with `cluster-admin` privileges.

**Procedure**

Create a namespace. For example:

```bash
$ oc new-project simple-kmod-demo
```

1. The YAML defines an ImageStream for storing the simple-kmod driver container image, and a BuildConfig for building the container. Save this YAML as `0000-buildconfig.yaml.template`.

```yaml
apiVersion: image.openshift.io/v1
kind: ImageStream
metadata:
  labels:
    app: simple-kmod-driver-container
  name: simple-kmod-driver-container
namespace: simple-kmod-demo
```
spec: {}
---
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  labels:
    app: simple-kmod-driver-build
    name: simple-kmod-driver-build
    namespace: simple-kmod-demo
spec:
  nodeSelector:
    node-role.kubernetes.io/worker: ""
  runPolicy: "Serial"
triggers:
- type: "ConfigChange"
- type: "ImageChange"
source:
  git:
    ref: "master"
    uri: "https://github.com/openshift-psap/kvc-simple-kmod.git"
  type: Git
  dockerfile: |
    FROM DRIVER_TOOLKIT_IMAGE
    WORKDIR /build/
    # Expecting kmod software version as an input to the build
    ARG KMODVER
    # Grab the software from upstream
    RUN git clone https://github.com/openshift-psap/simple-kmod.git
    WORKDIR simple-kmod
    # Build and install the module
    RUN make all KVER=$(rpm -q --qf "%{VERSION}-%{RELEASE}-%{ARCH}" kernel-core) KMODVER=${KMODVER} \&& make install KVER=$(rpm -q --qf "%{VERSION}-%{RELEASE}-%{ARCH}" kernel-core) KMODVER=${KMODVER}
    # Add the helper tools
    WORKDIR /root/kvc-simple-kmod
    ADD Makefile .
    ADD simple-kmod-lib.sh .
    ADD simple-kmod-wrapper.sh .
    ADD simple-kmod.conf .
    RUN mkdir -p /usr/lib/kvc/ \&& mkdir -p /etc/kvc/ \&&
    RUN systemctl enable kmods-via-containers@simple-kmod

strategy:
  dockerStrategy:
    buildArgs:
      - name: KMODVER
        value: DEMO
output:
2. Substitute the correct driver toolkit image for the OpenShift Container Platform version you are running in place of “DRIVER_TOOLKIT_IMAGE” with the following commands.

```
$ OCP_VERSION=$(oc get clusterversion/version -ojsonpath={.status.desired.version})
$ DRIVER_TOOLKIT_IMAGE=$(oc adm release info $OCP_VERSION --image-for=driver-toolkit)
$ sed "s#DRIVER_TOOLKIT_IMAGE#${DRIVER_TOOLKIT_IMAGE}#" 0000-buildconfig.yaml.template > 0000-buildconfig.yaml
```

3. Create the image stream and build config with

```
$ oc create -f 0000-buildconfig.yaml
```

4. After the builder pod completes successfully, deploy the driver container image as a DaemonSet.

a. The driver container must run with the privileged security context in order to load the kernel modules on the host. The following YAML file contains the RBAC rules and the DaemonSet for running the driver container. Save this YAML as `1000-drivercontainer.yaml`.

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: simple-kmod-driver-container
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: simple-kmod-driver-container
rules:
  - apiGroups:
    - security.openshift.io
    resources:
    - securitycontextconstraints
    verbs:
    - use
    resourceNames:
    - privileged
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: simple-kmod-driver-container
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: simple-kmod-driver-container
subjects:
Create the RBAC rules and daemon set:

```yaml
- kind: ServiceAccount
  name: simple-kmod-driver-container
  userNames:
- system:serviceaccount:simple-kmod-demo:simple-kmod-driver-container
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: simple-kmod-driver-container
spec:
  selector:
    matchLabels:
      app: simple-kmod-driver-container
  template:
    metadata:
      labels:
        app: simple-kmod-driver-container
    spec:
      serviceAccount: simple-kmod-driver-container
      serviceAccountName: simple-kmod-driver-container
      containers:
      - image: image-registry.openshift-image-registry.svc:5000/simple-kmod-demo/simple-kmod-driver-container:demo
        name: simple-kmod-driver-container
        imagePullPolicy: Always
        command: ["/sbin/init"]
        lifecycle:
          preStop:
            exec:
              command: ["/bin/sh", "-c", "systemctl stop kmods-via-containers@simple-kmod"]
        securityContext:
          privileged: true
        nodeSelector:
          node-role.kubernetes.io/worker: '
```

b. Create the RBAC rules and daemon set:

```
$ oc create -f 1000-drivercontainer.yaml
```

5. After the pods are running on the worker nodes, verify that the simple_kmod kernel module is loaded successfully on the host machines with `lsmod`.

a. Verify that the pods are running:

```
$ oc get pod -n simple-kmod-demo
```

**Example output**

```
NAME                                 READY   STATUS      RESTARTS   AGE
simple-kmod-driver-build-1-build     0/1     Completed   0          6m
simple-kmod-driver-container-b22fd   1/1     Running     0          40s
simple-kmod-driver-container-jz9vn   1/1     Running     0          40s
simple-kmod-driver-container-p45cc   1/1     Running     0          40s
```

b. Execute the `lsmod` command in the driver container pod:
$ oc exec -it pod/simple-kmod-driver-container-p45cc -- lsmod | grep simple

Example output

- simple_procfs_kmod 16384 0
- simple_kmod 16384 0

2.4. ADDITIONAL RESOURCES

- For more information about configuring registry storage for your cluster, see Image Registry Operator in OpenShift Container Platform.
CHAPTER 3. SPECIAL RESOURCE OPERATOR

Learn about the Special Resource Operator (SRO) and how you can use it to build and manage driver containers for loading kernel modules and device drivers on nodes in an OpenShift Container Platform cluster.

**IMPORTANT**

The Special Resource Operator is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see [https://access.redhat.com/support/offerings/techpreview/](https://access.redhat.com/support/offerings/techpreview/).

3.1. ABOUT THE SPECIAL RESOURCE OPERATOR

The Special Resource Operator (SRO) helps you manage the deployment of kernel modules and drivers on an existing OpenShift Container Platform cluster. The SRO can be used for a case as simple as building and loading a single kernel module, or as complex as deploying the driver, device plug-in, and monitoring stack for a hardware accelerator.

For loading kernel modules, the SRO is designed around the use of driver containers. Driver containers are increasingly being used in cloud-native environments, especially when run on pure container operating systems, to deliver hardware drivers to the host. Driver containers extend the kernel stack beyond the out-of-the-box software and hardware features of a specific kernel. Driver containers work on various container-capable Linux distributions. With driver containers, the host operating system stays clean and there is no clash between different library versions or binaries on the host.

**NOTE**

The functions described require a connected environment with a constant connection to the network. These functions are not available for disconnected environments.

3.2. INSTALLING THE SPECIAL RESOURCE OPERATOR

As a cluster administrator, you can install the Special Resource Operator (SRO) by using the OpenShift CLI or the web console.

3.2.1. Installing the Special Resource Operator by using the CLI

As a cluster administrator, you can install the Special Resource Operator (SRO) by using the OpenShift CLI.

**Prerequisites**

- You have a running OpenShift Container Platform cluster.
- You installed the OpenShift CLI (`oc`).
- You are logged into the OpenShift CLI as a user with **cluster-admin** privileges.
Procedure

1. Install the SRO in the openshift-operators namespace:
   a. Create the following Subscription CR and save the YAML in the sro-sub.yaml file:

   **Example Subscription CR**
   ```yaml
   apiVersion: operators.coreos.com/v1alpha1
   kind: Subscription
   metadata:
     name: openshift-special-resource-operator
     namespace: openshift-operators
   spec:
     channel: "stable"
     installPlanApproval: Automatic
     name: openshift-special-resource-operator
     source: redhat-operators
     sourceNamespace: openshift-marketplace
   ```

   b. Create the subscription object by running the following command:

   ```bash
   $ oc create -f sro-sub.yaml
   ```

   c. Switch to the openshift-operators project:

   ```bash
   $ oc project openshift-operators
   ```

Verification

- To verify that the Operator deployment is successful, run:

  ```bash
  $ oc get pods
  ```

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>nfd-controller-manager-7f4c5f5778-4lvvk</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>89s</td>
</tr>
<tr>
<td>special-resource-controller-manager-6dbf7d4f6f-9kl8h</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>81s</td>
</tr>
</tbody>
</table>

A successful deployment shows a **Running** status.

3.2.2. Installing the Special Resource Operator by using the web console

As a cluster administrator, you can install the Special Resource Operator (SRO) by using the OpenShift Container Platform web console.

Procedure

1. Log in to the OpenShift Container Platform web console.

2. Install the Special Resource Operator:

b. Choose Special Resource Operator from the list of available Operators, and then click Install.

c. On the Install Operator page, select a specific namespace on the cluster select the namespace created in the previous section, and then click Install.

Verification

To verify that the Special Resource Operator installed successfully:

1. Navigate to the Operators → Installed Operators page.

2. Ensure that Special Resource Operator is listed in the openshift-operators project with a Status of InstallSucceeded.

   **NOTE**

   During installation, an Operator might display a Failed status. If the installation later succeeds with an InstallSucceeded message, you can ignore the Failed message.

3. If the Operator does not appear as installed, to troubleshoot further:

   a. Navigate to the Operators → Installed Operators page and inspect the Operator Subscriptions and Install Plans tabs for any failure or errors under Status.

   b. Navigate to the Workloads → Pods page and check the logs for pods in the openshift-operators project.

### 3.3. USING THE SPECIAL RESOURCE OPERATOR

The Special Resource Operator (SRO) is used to manage the build and deployment of a driver container. The objects required to build and deploy the container can be defined in a Helm chart.

The example in this section uses the simple-kmod SpecialResource object to point to a ConfigMap object that is created to store the Helm charts.

#### 3.3.1. Building and running the simple-kmod SpecialResource by using a config map

In this example, the simple-kmod kernel module shows how the Special Resource Operator (SRO) manages a driver container. The container is defined in the Helm chart templates that are stored in a config map.

**Prerequisites**

- You have a running OpenShift Container Platform cluster.
- You set the Image Registry Operator state to Managed for your cluster.
- You installed the OpenShift CLI (oc).
- You are logged into the OpenShift CLI as a user with cluster-admin privileges.
- You installed the Node Feature Discovery (NFD) Operator.
You installed the SRO.

You installed the Helm CLI (helm).

Procedure

1. To create a simple-kmod SpecialResource object, define an image stream and build config to build the image, and a service account, role, role binding, and daemon set to run the container. The service account, role, and role binding are required to run the daemon set with the privileged security context so that the kernel module can be loaded.

   a. Create a templates directory, and change into it:

   ```
   $ mkdir -p chart/simple-kmod-0.0.1/templates
   $ cd chart/simple-kmod-0.0.1/templates
   ```

   b. Save this YAML template for the image stream and build config in the templates directory as 0000-buildconfig.yaml:

   ```yaml
   apiVersion: image.openshift.io/v1
   kind: ImageStream
   metadata:
     labels:
       app: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverContainer}}
       name: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverContainer}}
   spec: {}
   ---
   apiVersion: build.openshift.io/v1
   kind: BuildConfig
   metadata:
     labels:
       app: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverBuild}}
       name: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverBuild}}
     annotations:
       specialresource.openshift.io/wait: "true"
       specialresource.openshift.io/driver-container-vendor: simple-kmod
       specialresource.openshift.io/kernel-affine: "true"
   spec:
     nodeSelector:
       node-role.kubernetes.io/worker: ""
     runPolicy: "Serial"
     triggers:
       - type: "ConfigChange"
       - type: "ImageChange"
     source:
       git:
         ref: {{.Values.specialresource.spec.driverContainer.source.git.ref}}
         uri: {{.Values.specialresource.spec.driverContainer.source.git.uri}}
       type: Git
     strategy:
   ```
The templates such as `{{.Values.specialresource.metadata.name}}` are filled in by the SRO, based on fields in the `SpecialResource` CR and variables known to the Operator such as `{{.Values.KernelFullVersion}}`.

c. Save the following YAML template for the RBAC resources and daemon set in the `templates` directory as `1000-driver-container.yaml`:

```yaml
dockerStrategy:
dockerfilePath: Dockerfile.SRO
buildArgs:
  - name: "IMAGE"
    value: {{ .Values.driverToolkitImage }}
  {{- range $arg := .Values.buildArgs }}
    - name: {{ $arg.name }}
      value: {{ $arg.value }}
  {{- end }}
  - name: KVER
    value: {{ .Values.kernelFullVersion }}
output:
to:
  kind: ImageStreamTag
  name: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverContainer}}:v{{.Values.kernelFullVersion}}
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverContainer}}
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverContainer}}
rules:
  - apiGroups:
    - security.openshift.io
    resources:
      - securitycontextconstraints
    verbs:
      - use
        resourceNames:
        - privileged
    ---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverContainer}}
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverContainer}}
```
subjects:  
- kind: ServiceAccount  
  name: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverContainer}}  
  namespace: {{.Values.specialresource.spec.namespace}}

---

apiVersion: apps/v1  
kind: DaemonSet  
metadata:  
  labels:  
    app: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverContainer}}  
    name: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverContainer}}  
  annotations:  
    specialresource.openshift.io/wait: "true"  
    specialresource.openshift.io/state: "driver-container"  
    specialresource.openshift.io/driver-container-vendor: simple-kmod  
    specialresource.openshift.io/kernel-affine: "true"  
    specialresource.openshift.io/from-configmap: "true"

spec:  
  updateStrategy:  
    type: OnDelete  
  selector:  
    matchLabels:  
      app: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverContainer}}  
  template:  
    metadata:  
      # Mark this pod as a critical add-on; when enabled, the critical add-on scheduler reserves resources for critical add-on pods so that they can be rescheduled after a failure. This annotation works in tandem with the toleration below.  
      annotations:  
        scheduler.alpha.kubernetes.io/critical-pod: ""  
      labels:  
        app: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverContainer}}

spec:  
  serviceAccount: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverContainer}}  
  serviceAccountName: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverContainer}}

containers:  
- image: image-registry.openshift-image-registry.svc:5000/{{.Values.specialresource.spec.namespace}}/{{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverContainer}}:v{{.Values.kernelFullVersion}}  
  name: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverContainer}}

  imagePullPolicy: Always  
  command: ["/sbin/init"]  
  lifecycle:  
    preStop:  
      exec:  
        command: ["/bin/sh", "-c", "systemctl stop kmods-via-containers@{{.Values.specialresource.metadata.name}}"]

  securityContext:
Change into the `chart/simple-kmod-0.0.1` directory:

```bash
$ cd ..
```

Save the following YAML for the chart as `Chart.yaml` in the `chart/simple-kmod-0.0.1` directory:

```yaml
apiVersion: v2
category: simple-kmod
description: Simple kmod will deploy a simple kmod driver-container
icon: https://avatars.githubusercontent.com/u/55542927
type: application
version: 0.0.1
appVersion: 1.0.0
```

2. From the `chart` directory, create the chart using the `helm package` command:

```bash
$ helm package simple-kmod-0.0.1/
```

**Example output**

Successfully packaged chart and saved it to:

```
/data/<username>/git/<github_username>/special-resource-operator/docs/chart/simple-kmod-0.0.1/simple-kmod-0.0.1.tgz
```

3. Create a config map to store the chart files:

   a. Create a directory for the config map files:

      ```bash
      $ mkdir cm
      ```

   b. Copy the Helm chart into the `cm` directory:

      ```bash
      $ cp simple-kmod-0.0.1.tgz cm/simple-kmod-0.0.1.tgz
      ```

   c. Create an index file specifying the Helm repo that contains the Helm chart:

      ```bash
      $ helm repo index cm --url=cm://simple-kmod/simple-kmod-chart
      ```

   d. Create a namespace for the objects defined in the Helm chart:

      ```bash
      $ oc create namespace simple-kmod
      ```

   e. Create the config map object:

      ```bash
      $ oc create cm simple-kmod-chart --from-file=cm/index.yaml --from-file=cm/simple-kmod-0.0.1.tgz -n simple-kmod
      ```
4. Use the following **SpecialResource** manifest to deploy the simple-kmod object using the Helm chart that you created in the config map. Save this YAML as `simple-kmod-configmap.yaml`:

```yaml
apiVersion: sro.openshift.io/v1beta1
class: SpecialResource
metadata:
  name: simple-kmod
spec:
  #debug: true
  namespace: simple-kmod
deployment:
  chart:
    name: simple-kmod
    version: 0.0.1
    repository:
      name: example
      url: cm://simple-kmod/simple-kmod-chart
  set:
    kind: Values
    apiVersion: sro.openshift.io/v1beta1
    kmodNames: 
      - simple-kmod
      - simple-procfs-kmod
    buildArgs:
      - name: "KMODVER"
        value: "SRO"
driverContainer:
  source:
    git:
      ref: "master"
      uri: "https://github.com/openshift-psap/kvc-simple-kmod.git"
```

1. Optional: Uncomment the `#debug: true` line to have the YAML files in the chart printed in full in the Operator logs and to verify that the logs are created and templated properly.

2. The `spec.chart.repository.url` field tells the SRO to look for the chart in a config map.

5. From a command line, create the **SpecialResource** file:

```
$ oc create -f simple-kmod-configmap.yaml
```

**NOTE**

To remove the simple-kmod kernel module from the node, delete the simple-kmod **SpecialResource** API object using the `oc delete` command. The kernel module is unloaded when the driver container pod is deleted.

**Verification**

The **simple-kmod** resources are deployed in the **simple-kmod** namespace as specified in the object manifest. After a short time, the build pod for the **simple-kmod** driver container starts running. The build completes after a few minutes, and then the driver container pods start running.

1. Use `oc get pods` command to display the status of the build pods:

```
$ oc get pods -n simple-kmod
```
Example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple-kmod-driver-build-12813789169ac0ee-1-build</td>
<td>0/1</td>
<td>Completed</td>
<td>0</td>
<td>7m12s</td>
</tr>
<tr>
<td>simple-kmod-driver-container-12813789169ac0ee-mjsnh</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>8m2s</td>
</tr>
<tr>
<td>simple-kmod-driver-container-12813789169ac0ee-qtkff</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>8m2s</td>
</tr>
</tbody>
</table>

2. Use the `oc logs` command, along with the build pod name obtained from the `oc get pods` command above, to display the logs of the simple-kmod driver container image build:

   `$ oc logs pod/simple-kmod-driver-build-12813789169ac0ee-1-build -n simple-kmod`

3. To verify that the simple-kmod kernel modules are loaded, execute the `lsmod` command in one of the driver container pods that was returned from the `oc get pods` command above:

   `$ oc exec -n simple-kmod -it pod/simple-kmod-driver-container-12813789169ac0ee-mjsnh -- lsmod | grep simple`

Example output

| simple_procfs_kmod     | 16384 | 0       |
| simple_kmod            | 16384 | 0       |

TIP

The `sro_kind_completed_info` SRO Prometheus metric provides information about the status of the different objects being deployed, which can be useful to troubleshoot SRO CR installations. The SRO also provides other types of metrics that you can use to watch the health of your environment.

3.3.2. Building and running the simple-kmod SpecialResource for a hub-and-spoke topology

You can use the Special Resource Operator (SRO) on a hub-and-spoke deployment in which Red Hat Advanced Cluster Management (RHACM) connects a hub cluster to one or more managed clusters.

This example procedure shows how the SRO builds driver containers in the hub. The SRO watches hub cluster resources to identify OpenShift Container Platform versions for the helm charts that it uses to create resources which it delivers to spokes.

Prerequisites

- You have a running OpenShift Container Platform cluster.
- You installed the OpenShift CLI (`oc`).
- You are logged into the OpenShift CLI as a user with `cluster-admin` privileges.
- You installed the SRO.
- You installed the Helm CLI (`helm`).
- You installed Red Hat Advanced Cluster Management (RHACM).
You configured a container registry.

Procedure

1. Create a templates directory by running the following command:

   ```bash
   $ mkdir -p charts/acm-simple-kmod-0.0.1/templates
   ```

2. Change to the templates directory by running the following command:

   ```bash
   $ cd charts/acm-simple-kmod-0.0.1/templates
   ```

3. Create templates files for the BuildConfig, Policy, and PlacementRule resources.
   a. Save this YAML template for the image stream and build config in the templates directory as `0001-buildconfig.yaml`.

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  labels:
    app: {{ printf "%s-%s" .Values.specialResourceModule.metadata.name .Values.kernelFullVersion | replace "." "-" | replace "_" "-" | trunc 63 }}
    name: {{ printf "%s-%s" .Values.specialResourceModule.metadata.name .Values.kernelFullVersion | replace "." "-" | replace "_" "-" | trunc 63 }}
  annotations:
    specialresource.openshift.io/wait: "true"
spec:
  nodeSelector:
    node-role.kubernetes.io/worker: ""
  runPolicy: "Serial"
  triggers:
    - type: "ConfigChange"
    - type: "ImageChange"
  source:
    dockerfile: |
    FROM {{ .Values.driverToolkitImage } } as builder
    WORKDIR /build/
    RUN git clone -b {{.Values.specialResourceModule.spec.set.git.ref}}
    {{.Values.specialResourceModule.spec.set.git.uri}}
    WORKDIR /build/simple-kmod
    RUN make all install KVER={{ .Values.kernelFullVersion }}
    FROM registry.redhat.io/ubi8/ubi-minimal
    RUN microdnf -y install kmod
    COPY --from=builder /etc/driver-toolkit-release.json /etc/
    COPY --from=builder /lib/modules/{{ .Values.kernelFullVersion }}/ /lib/modules/{}
  strategy:
    dockerfileStrategy:
      dockerfilePath: Dockerfile.SRO
      buildArgs:
        - name: "IMAGE"
          value: {{ .Values.driverToolkitImage }}
        {{- range $arg := .Values.buildArgs }}
        - name: {{ $arg.name }}
          value: {{ .Values.buildArgs }}
        {{- end }}
```

```
Save this YAML template for the ACM policy in the templates directory as 0002-policy.yaml.

```yaml
apiVersion: policy.open-cluster-management.io/v1
kind: Policy
metadata:
  name: policy-{{.Values.specialResourceModule.metadata.name}}-ds
annotations:
  policy.open-cluster-management.io/categories: CM Configuration Management
  policy.open-cluster-management.io/controls: CM-2 Baseline Configuration
  policy.open-cluster-management.io/standards: NIST-CSF
spec:
  remediationAction: enforce
  disabled: false
policy-templates:
  - objectDefinition:
      apiVersion: policy.open-cluster-management.io/v1
      kind: ConfigurationPolicy
      metadata:
        name: config-{{.Values.specialResourceModule.metadata.name}}-ds
      spec:
        remediationAction: enforce
        severity: low
        namespacesSelector:
          exclude:
          - kube-*
          include:
            - **
    object-templates:
      - complianceType: musthave
        objectDefinition:
          apiVersion: v1
          kind: Namespace
          metadata:
            name: {{.Values.specialResourceModule.spec.namespace}}
      - complianceType: mustonlyhave
        objectDefinition:
          apiVersion: v1
          kind: ServiceAccount
          metadata:
            name: {{.Values.specialResourceModule.metadata.name}}
            namespace: {{.Values.specialResourceModule.spec.namespace}}
      - complianceType: mustonlyhave
        objectDefinition:
          apiVersion: rbac.authorization.k8s.io/v1
          kind: Role
```

b. Save this YAML template for the ACM policy in the templates directory as 0002-policy.yaml.
metadata:
  name: {{.Values.specialResourceModule.metadata.name}}
  namespace: {{.Values.specialResourceModule.spec.namespace}}
rules:
  - apiGroups:
    - security.openshift.io
  resources:
    - securitycontextconstraints
    verbs:
    - use
  resourceNames:
    - privileged
  - complianceType: mustonlyhave
objectDefinition:
  apiVersion: rbac.authorization.k8s.io/v1
  kind: RoleBinding
  metadata:
    name: {{.Values.specialResourceModule.metadata.name}}
    namespace: {{.Values.specialResourceModule.spec.namespace}}
  roleRef:
    apiGroup: rbac.authorization.k8s.io
    kind: Role
    name: {{.Values.specialResourceModule.metadata.name}}
  subjects:
    - kind: ServiceAccount
      name: {{.Values.specialResourceModule.metadata.name}}
      namespace: {{.Values.specialResourceModule.spec.namespace}}
      complianceType: musthave
objectDefinition:
  apiVersion: apps/v1
  kind: DaemonSet
  metadata:
    labels:
      app: {{ printf "%s-%s" .Values.specialResourceModule.metadata.name .Values.kernelFullVersion | replace "." "-" | replace "_" "-" | trunc 63 }}
    name: {{ printf "%s-%s" .Values.specialResourceModule.metadata.name .Values.kernelFullVersion | replace "." "-" | replace "_" "-" | trunc 63 }}
    namespace: {{.Values.specialResourceModule.spec.namespace}}
  spec:
    updateStrategy:
      type: OnDelete
    selector:
      matchLabels:
        app: {{ printf "%s-%s" .Values.specialResourceModule.metadata.name .Values.kernelFullVersion | replace "." "-" | replace "_" "-" | trunc 63 }}
    template:
      metadata:
        annotations:
          scheduler.alpha.kubernetes.io/critical-pod: ""
Save this YAML template for the placement of policies in the templates directory as 0003-policy.yaml.

```yaml
apiVersion: apps.open-cluster-management.io/v1
kind: PlacementRule
metadata:
  name: {{.Values.specialResourceModule.metadata.name}}-placement
spec:
  clusterConditions:
    - status: "True"
      type: ManagedClusterConditionAvailable
  clusterSelector:
    matchExpressions:
      - key: name
        operator: NotIn
        values:
        - local-cluster
---
apiVersion: policy.open-cluster-management.io/v1
kind: PlacementBinding
metadata:
  name: {{.Values.specialResourceModule.metadata.metadata.name}}-binding
placementRef:
  apiGroup: apps.open-cluster-management.io
  kind: PlacementRule
  name: {{.Values.specialResourceModule.metadata.name}}-placement
subjects:
  - apiGroup: policy.open-cluster-management.io
    kind: Policy
    name: policy-{{.Values.specialResourceModule.metadata.name}}-ds
```

c. Save this YAML template for the placement of policies in the templates directory as 0003-policy.yaml.

```yaml
d. Change into the charts/acm-simple-kmod-0.0.1 directory by running the following command:

    cd ..
```
Save the following YAML template for the chart as `Chart.yaml` in the `charts/acm-simple-kmod-0.0.1` directory:

```yaml
apiVersion: v2
name: acm-simple-kmod
description: Build ACM enabled simple-kmod driver with SpecialResourceOperator
icon: https://avatars.githubusercontent.com/u/55542927
type: application
version: 0.0.1
appVersion: 1.6.4
```

4. From the charts directory, create the chart using the command:

```
$ helm package acm-simple-kmod-0.0.1/
```

Example output

```
Successfully packaged chart and saved it to: <directory>/charts/acm-simple-kmod-0.0.1.tgz
```

5. Create a config map to store the chart files.

   a. Create a directory for the config map files by running the following command:

```
$ mkdir cm
```

   b. Copy the Helm chart into the cm directory by running the following command:

```
$ cp acm-simple-kmod-0.0.1.tgz cm/acm-simple-kmod-0.0.1.tgz
```

   c. Create an index file specifying the Helm repository that contains the Helm chart by running the following command:

```
$ helm repo index cm --url=cm://acm-simple-kmod/acm-simple-kmod-chart
```

   d. Create a namespace for the objects defined in the Helm chart by running the following command:

```
$ oc create namespace acm-simple-kmod
```

   e. Create the config map object by running the following command:

```
$ oc create cm acm-simple-kmod-chart --from-file=cm/index.yaml --from-file=cm/acm-simple-kmod-0.0.1.tgz -n acm-simple-kmod
```

6. Use the following SpecialResourceModule manifest to deploy the simple-kmod object using the Helm chart that you created in the config map. Save this YAML file as `acm-simple-kmod.yaml`:

```yaml
apiVersion: sro.openshift.io/v1beta1
kind: SpecialResourceModule
metadata:
  name: acm-simple-kmod
spec:
```

---

25
namespace: acm-simple-kmod
chart:
  name: acm-simple-kmod
  version: 0.0.1
repository:
  name: acm-simple-kmod
  url: cm://acm-simple-kmod/acm-simple-kmod-chart
set:
  kind: Values
  apiVersion: sro.openshift.io/v1beta1
  buildArgs:
    - name: "KMODVER"
      value: "SRO"
  registry: <your_registry> 1
  git:
    ref: master
    uri: https://github.com/openshift-psap/kvc-simple-kmod.git
  watch:
    - path: "$metadata.labels.openshiftVersion"
      apiVersion: cluster.open-cluster-management.io/v1
      kind: ManagedCluster
      name: spoke1

Specify the URL for a registry that you have configured.

7. Create the special resource module by running the following command:

   $ oc apply -f charts/examples/acm-simple-kmod.yaml

Verification

1. Check the status of the build pods by running the following command:

   $ KUBECONFIG=~/.hub/auth/kubeconfig oc get pod -n acm-simple-kmod

Example output

   NAME                                                   READY   STATUS      RESTARTS   AGE
   acm-simple-kmod-4-18-0-305-34-2-el8-4-x86-64-1-build   0/1     Completed   0          42m

2. Check that the policies have been created by running the following command:

   $ KUBECONFIG=~/.hub/auth/kubeconfig oc get placementrules,placementbindings,policies -n acm-simple-kmod

Example output

   NAME                                                                 AGE   REPLICAS
   placementrule.apps.open-cluster-management.io/acm-simple-kmod-placement   40m
   placementbinding.policy.open-cluster-management.io/acm-simple-kmod-binding   40m
Check that the resources have been reconciled by running the following command:

```bash
$ KUBECONFIG=~/.hub/auth/kubeconfig oc get specialresourcemodule acm-simple-kmod -o json | jq -r '.status'
```

Example output

```json
{
  "versions": {
    "quay.io/openshift-release-dev/ocp-v4.0-art-dev@sha256:6a3330ef5a178435721ff4efdde762261a9c55212e9b4534385e04037693f8e4": {
      "complete": true
    }
  }
}
```

Check that the resources are running in the spoke by running the following command:

```bash
$ KUBECONFIG=~/.spoke1/kubeconfig oc get ds,pod -n acm-simple-kmod
```

Example output

```
NAME                                                        READY   STATUS    RESTARTS   AGE
pod/acm-simple-kmod-4-18-0-305-45-1-el8-4-x86-64-brw78       1/1     Running   0          26m
pod/acm-simple-kmod-4-18-0-305-45-1-el8-4-x86-64-fqh5h        1/1     Running   0          26m
pod/acm-simple-kmod-4-18-0-305-45-1-el8-4-x86-64-m9sfd       1/1     Running   0          26m
```

### 3.4. PROMETHEUS SPECIAL RESOURCE OPERATOR METRICS

The Special Resource Operator (SRO) exposes the following Prometheus metrics through the **metrics** service:

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sro_used_nodes</td>
<td>Returns the nodes that are running pods created by a SRO custom resource (CR). This metric is available for DaemonSet and Deployment objects only.</td>
</tr>
</tbody>
</table>
### Metric Name

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sro_kind_completed_info</td>
<td>Represents whether a kind of an object defined by the Helm Charts in a SRO CR has been successfully uploaded in the cluster (value 1) or not (value 0). Examples of objects are DaemonSet, Deployment or BuildConfig.</td>
</tr>
<tr>
<td>sro_states_completed_info</td>
<td>Represents whether the SRO has finished processing a CR successfully (value 1) or the SRO has not processed the CR yet (value 0).</td>
</tr>
<tr>
<td>sro_managed_resources_total</td>
<td>Returns the number of SRO CRs in the cluster, regardless of their state.</td>
</tr>
</tbody>
</table>

### 3.5. ADDITIONAL RESOURCES

- For information about restoring the Image Registry Operator state before using the Special Resource Operator, see [Image registry removed during installation](#).

- For details about installing the NFD Operator see [Node Feature Discovery (NFD) Operator](#).
CHAPTER 4. NODE FEATURE DISCOVERY OPERATOR

Learn about the Node Feature Discovery (NFD) Operator and how you can use it to expose node-level information by orchestrating Node Feature Discovery, a Kubernetes add-on for detecting hardware features and system configuration.

4.1. ABOUT THE NODE FEATURE DISCOVERY OPERATOR

The Node Feature Discovery Operator (NFD) manages the detection of hardware features and configuration in an OpenShift Container Platform cluster by labeling the nodes with hardware-specific information. NFD labels the host with node-specific attributes, such as PCI cards, kernel, operating system version, and so on.

The NFD Operator can be found on the Operator Hub by searching for "Node Feature Discovery".

4.2. INSTALLING THE NODE FEATURE DISCOVERY OPERATOR

The Node Feature Discovery (NFD) Operator orchestrates all resources needed to run the NFD daemon set. As a cluster administrator, you can install the NFD Operator by using the OpenShift Container Platform CLI or the web console.

4.2.1. Installing the NFD Operator using the CLI

As a cluster administrator, you can install the NFD Operator using the CLI.

Prerequisites

- An OpenShift Container Platform cluster
- Install the OpenShift CLI (`oc`).
- Log in as a user with `cluster-admin` privileges.

Procedure

1. Create a namespace for the NFD Operator.
   
   a. Create the following `Namespace` custom resource (CR) that defines the `openshift-nfd` namespace, and then save the YAML in the `nfd-namespace.yaml` file:

   ```yaml
   apiVersion: v1
   kind: Namespace
   metadata:
     name: openshift-nfd
   ```

   b. Create the namespace by running the following command:

   ```bash
   $ oc create -f nfd-namespace.yaml
   ```

2. Install the NFD Operator in the namespace you created in the previous step by creating the following objects:

   a. Create the following `OperatorGroup` CR and save the YAML in the `nfd-operatorgroup.yaml` file:
Create the OperatorGroup CR by running the following command:

```
$ oc create -f nfd-operatorgroup.yaml
```

c. Create the following Subscription CR and save the YAML in the `nfd-sub.yaml` file:

**Example Subscription**

```yaml
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: nfd
  namespace: openshift-nfd
spec:
  channel: "stable"
  installPlanApproval: Automatic
  name: nfd
  source:
    name: redhat-operators
    sourceNamespace: openshift-marketplace
```

d. Create the subscription object by running the following command:

```
$ oc create -f nfd-sub.yaml
```

e. Change to the `openshift-nfd` project:

```
$ oc project openshift-nfd
```

**Verification**

- To verify that the Operator deployment is successful, run:

```
$ oc get pods
```

**Example output**

```
NAME                                      READY   STATUS    RESTARTS   AGE
nfd-controller-manager-7f86ccfb58-vgr4x   2/2     Running   0          10m
```

A successful deployment shows a **Running** status.

### 4.2.2. Installing the NFD Operator using the web console
As a cluster administrator, you can install the NFD Operator using the web console.

Procedure

1. In the OpenShift Container Platform web console, click Operators → OperatorHub.
2. Choose Node Feature Discovery from the list of available Operators, and then click Install.
3. On the Install Operator page, select A specific namespace on the cluster and then click Install. You do not need to create a namespace because it is created for you.

Verification

To verify that the NFD Operator installed successfully:

1. Navigate to the Operators → Installed Operators page.
2. Ensure that Node Feature Discovery is listed in the openshift-nfd project with a Status of InstallSucceeded.

NOTE
During installation an Operator might display a Failed status. If the installation later succeeds with an InstallSucceeded message, you can ignore the Failed message.

Troubleshooting

If the Operator does not appear as installed, troubleshoot further:

1. Navigate to the Operators → Installed Operators page and inspect the Operator Subscriptions and Install Plans tabs for any failure or errors under Status.
2. Navigate to the Workloads → Pods page and check the logs for pods in the openshift-nfd project.

4.3. USING THE NODE FEATURE DISCOVERY OPERATOR

The Node Feature Discovery (NFD) Operator orchestrates all resources needed to run the Node-Feature-Discovery daemon set by watching for a NodeFeatureDiscovery CR. Based on the NodeFeatureDiscovery CR, the Operator will create the operand (NFD) components in the desired namespace. You can edit the CR to choose another namespace, image, imagePullPolicy, and nfd-worker-conf, among other options.

As a cluster administrator, you can create a NodeFeatureDiscovery instance using the OpenShift Container Platform CLI or the web console.

4.3.1. Create a NodeFeatureDiscovery instance using the CLI

As a cluster administrator, you can create a NodeFeatureDiscovery CR instance using the CLI.

Prerequisites

- An OpenShift Container Platform cluster
Install the OpenShift CLI (oc).

Log in as a user with cluster-admin privileges.

Install the NFD Operator.

Procedure

1. Create the following NodeFeatureDiscovery Custom Resource (CR), and then save the YAML in the NodeFeatureDiscovery.yaml file:

```yaml
apiVersion: nfd.openshift.io/v1
kind: NodeFeatureDiscovery
metadata:
  name: nfd-instance
  namespace: openshift-nfd
spec:
  instance: "" # instance is empty by default
topologyupdater: false # False by default
operand:
  imagePullPolicy: Always
workerConfig:
  configData:
    | core:
    # labelWhiteList:
    # noPublish: false
    sleepInterval: 60s
    # sources: [all]
    # klog:
    #  addDirHeader: false
    #  alsoLogToStderr: false
    #  logBacktraceAt:
    #  logToStderr: true
    #  skipHeaders: false
    #  stderrThreshold: 2
    #  v: 0
    #  vmodule:
    ## NOTE: the following options are not dynamically run-time configurable
    ## and require a nfd-worker restart to take effect after being changed
    #  logDir:
    #  logFile:
    #  logFileSize: 1800
    #  skipLogHeaders: false
  sources:
    cpu:
      cpuid:
        # NOTE: whitelist has priority over blacklist
        attributeBlacklist:
          - "BMI1"
          - "BMI2"
          - "CLMUL"
          - "CMOV"
          - "CX16"
          - "ERMS"
          - "F16C"
```
For more details on how to customize NFD workers, refer to the Configuration file reference of nfd-worker.

1. Create the `NodeFeatureDiscovery` CR instance by running the following command:

   ```bash
   $ oc create -f NodeFeatureDiscovery.yaml
   ``

Verification

- To verify that the instance is created, run:

  ```bash
  $ oc get pods
  ```

Example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>nfd-controller-manager-7f86ccfb58-vgr4x</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>11m</td>
</tr>
<tr>
<td>nfd-master-hcn64</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>60s</td>
</tr>
</tbody>
</table>
A successful deployment shows a Running status.

4.3.2. Create a NodeFeatureDiscovery CR using the web console

Procedure

1. Navigate to the Operators → Installed Operators page.
2. Find Node Feature Discovery and see a box under Provided APIs.
3. Click Create instance.
4. Edit the values of the NodeFeatureDiscovery CR.
5. Click Create.

4.4. CONFIGURING THE NODE FEATURE DISCOVERY OPERATOR

4.4.1. core

The core section contains common configuration settings that are not specific to any particular feature source.

core.sleepInterval

core.sleepInterval specifies the interval between consecutive passes of feature detection or re-detection, and thus also the interval between node re-labeling. A non-positive value implies infinite sleep interval; no re-detection or re-labeling is done.

This value is overridden by the deprecated --sleep-interval command line flag, if specified.

Example usage

```
core:
  sleepInterval: 60s
```

The default value is 60s.

core.sources

core.sources specifies the list of enabled feature sources. A special value all enables all feature sources.

This value is overridden by the deprecated --sources command line flag, if specified.

Default: [all]

Example usage

```
core:
  sources:
```
core.labelWhiteList

*core.labelWhiteList* specifies a regular expression for filtering feature labels based on the label name. Non-matching labels are not published.

The regular expression is only matched against the basename part of the label, the part of the name after ‘/’. The label prefix, or namespace, is omitted.

This value is overridden by the deprecated *--label-whitelist* command line flag, if specified.

Default: *null*

**Example usage**

```
core:
  labelWhiteList: '^cpu-cpuid'
```

core.noPublish

Setting *core.noPublish* to *true* disables all communication with the *nfd-master*. It is effectively a dry run flag; *nfd-worker* runs feature detection normally, but no labeling requests are sent to *nfd-master*.

This value is overridden by the *--no-publish* command line flag, if specified.

**Example:**

**Example usage**

```
core:
  noPublish: true
```

The default value is *false*.

**core.klog**

The following options specify the logger configuration, most of which can be dynamically adjusted at run-time.

The logger options can also be specified using command line flags, which take precedence over any corresponding config file options.

**core.klog.addDirHeader**

If set to *true*, *core.klog.addDirHeader* adds the file directory to the header of the log messages.

Default: *false*

Run-time configurable: yes

**core.klog.alsologtostderr**

Log to standard error as well as files.

Default: *false*

Run-time configurable: yes

**core.klog.logBacktraceAt**
When logging hits line file:N, emit a stack trace.

Default: *empty*
Run-time configurable: yes

*core.klog.logDir*
If non-empty, write log files in this directory.

Default: *empty*
Run-time configurable: no

*core.klog.logFile*
If not empty, use this log file.

Default: *empty*
Run-time configurable: no

*core.klog.logFileMaxSize*
*core.klog.logFileMaxSize* defines the maximum size a log file can grow to. Unit is megabytes. If the value is 0, the maximum file size is unlimited.

Default: 1800
Run-time configurable: no

*core.klog.logtostderr*
Log to standard error instead of files

Default: true
Run-time configurable: yes

*core.klog.skipHeaders*
If *core.klog.skipHeaders* is set to true, avoid header prefixes in the log messages.

Default: false
Run-time configurable: yes

*core.klog.skipLogHeaders*
If *core.klog.skipLogHeaders* is set to true, avoid headers when opening log files.

Default: false
Run-time configurable: no

*core.klog.stderrthreshold*
Logs at or above this threshold go to stderr.

Default: 2
Run-time configurable: yes

*core.klog.v*
*core.klog.v* is the number for the log level verbosity.
Default: 0

Run-time configurable: yes

**core.klog.vmodule**

*core.klog.vmodule* is a comma-separated list of `pattern=N` settings for file-filtered logging.

Default: empty

Run-time configurable: yes

### 4.4.2. sources

The *sources* section contains feature source specific configuration parameters.

**sources.cpu.cpuid.attributeBlacklist**

Prevent publishing *cpuid* features listed in this option.

This value is overridden by *sources.cpu.cpuid.attributeWhitelist*, if specified.

Default: [BMI1, BMI2, CLMUL, CMOV, CX16, ERMS, F16C, HTT, L2CNT, MMX, MMXEXT, NX, POVCNT, RDRAND, RDSEED, RDTSCP, SGX, SGXLC, SSE, SSE2, SSE3, SSE4.1, SSE4.2, SSSE3]

**Example usage**

```
sources:
cpu:
  cpuid:
    attributeBlacklist: [MMX, MMXEXT]
```

**sources.cpu.cpuid.attributeWhitelist**

Only publish the *cpuid* features listed in this option.

**sources.cpu.cpuid.attributeWhitelist** takes precedence over *sources.cpu.cpuid.attributeBlacklist*.

Default: empty

**Example usage**

```
sources:
cpu:
  cpuid:
    attributeWhitelist: [AVX512BW, AVX512CD, AVX512DQ, AVX512F, AVX512VL]
```

**sources.kernel.kconfigFile**

*sources.kernel.kconfigFile* is the path of the kernel config file. If empty, NFD runs a search in the well-known standard locations.

Default: empty

**Example usage**

```
sources:
  kernel:
    kconfigFile: "/path/to/kconfig"
```
sources.kernel.configOpts
represents kernel configuration options to publish as feature labels.

Default: [NO_HZ, NO_HZ_IDLE, NO_HZ_FULL, PREEMPT]

Example usage
sources:
  kernel:
    configOpts: [NO_HZ, X86, DMI]

sources.pci.deviceClassWhitelist
is a list of PCI device class IDs for which to publish a label. It can be specified as a main class only (for example, 03) or full class-subclass combination (for example 0300). The former implies that all subclasses are accepted. The format of the labels can be further configured with deviceLabelFields.

Default: ["03", "0b40", "12"]

Example usage
sources:
  pci:
    deviceClassWhitelist: ["0200", "03"]

sources.pci.deviceLabelFields
is the set of PCI ID fields to use when constructing the name of the feature label. Valid fields are class, vendor, device, subsystem_vendor and subsystem_device.

Default: [class, vendor]

Example usage
sources:
  pci:
    deviceLabelFields: [class, vendor, device]

With the example config above, NFD would publish labels such as feature.node.kubernetes.io/pci-
<class-id>_<vendor-id>_<device-id>.present=true

sources.usb.deviceClassWhitelist
is a list of USB device class IDs for which to publish a feature label. The format of the labels can be further configured with deviceLabelFields.

Default: ["0e", "ef", "fe", "ff"]

Example usage
sources:
  usb:
    deviceClassWhitelist: ["ef", "ff"]
**sources.usb.deviceLabelFields** is the set of USB ID fields from which to compose the name of the feature label. Valid fields are **class**, **vendor**, and **device**.

Default: [class, vendor, device]

**Example usage**

```yaml
sources:
  pci:
    deviceLabelFields: [class, vendor]
```

With the example config above, NFD would publish labels like: `feature.node.kubernetes.io/usb-class-id_<vendor-id>.present=true`.

**sources.custom**

**sources.custom** is the list of rules to process in the custom feature source to create user-specific labels.

Default: empty

**Example usage**

```yaml
source:
  custom:
    - name: "my.custom.feature"
      matchOn:
        - loadedKMod: ["e1000e"]
        - pcid:
          class: ["0200"]
          vendor: ["8086"]
```

### 4.5. USING THE NFD TOPOLOGY UPDATER

The Node Feature Discovery (NFD) Topology Updater is a daemon responsible for examining allocated resources on a worker node. It accounts for resources that are available to be allocated to new pod on a per-zone basis, where a zone can be a Non-Uniform Memory Access (NUMA) node. The NFD Topology Updater communicates the information to nfd-master, which creates a **NodeResourceTopology** custom resource (CR) corresponding to all of the worker nodes in the cluster. One instance of the NFD Topology Updater runs on each node of the cluster.

To enable the Topology Updater workers in NFD, set the **topologyupdater** variable to **true** in the **NodeFeatureDiscovery** CR, as described in the section **Using the Node Feature Discovery Operator**.

#### 4.5.1. NodeResourceTopology CR

When run with NFD Topology Updater, NFD creates custom resource instances corresponding to the node resource hardware topology, such as:

```yaml
apiVersion: topology.node.k8s.io/v1alpha1
kind: NodeResourceTopology
metadata:
  name: node1
topologyPolicies: ["SingleNUMANodeContainerLevel"]
zones:
```

---

39
4.5.2. NFD Topology Updater command line flags

To view available command line flags, run the `nfd-topology-updater -help` command. For example, in a podman container, run the following command:

```
$ podman run gcr.io/k8s-staging-nfd/node-feature-discovery:master nfd-topology-updater -help
```

- **-ca-file**
The `-ca-file` flag is one of the three flags, together with the `-cert-file` and `-key-file` flags, that controls the mutual TLS authentication on the NFD Topology Updater. This flag specifies the TLS root certificate that is used for verifying the authenticity of nfd-master.

Default: empty

**IMPORTANT**
The `-ca-file` flag must be specified together with the `-cert-file` and `-key-file` flags.

Example
The `-cert-file` flag is one of the three flags, together with the `-ca-file` and `-key-file` flags, that controls mutual TLS authentication on the NFD Topology Updater. This flag specifies the TLS certificate presented for authenticating outgoing requests.

Default: empty

**IMPORTANT**

The `-cert-file` flag must be specified together with the `-ca-file` and `-key-file` flags.

**Example**

```bash
$ nfd-topology-updater -ca-file=/opt/nfd/ca.crt -cert-file=/opt/nfd/updater.crt -key-file=/opt/nfd/updater.key
```

The `-key-file` flag is one of the three flags, together with the `-ca-file` and `-cert-file` flags, that controls the mutual TLS authentication on the NFD Topology Updater. This flag specifies the private key corresponding the given certificate file, or `-cert-file`, that is used for authenticating outgoing requests.

Default: empty

**IMPORTANT**

The `-key-file` flag must be specified together with the `-ca-file` and `-cert-file` flags.

**Example**

```bash
$ nfd-topology-updater -ca-file=/opt/nfd/ca.crt -cert-file=/opt/nfd/updater.crt -key-file=/opt/nfd/updater.key
```

The `-kubelet-config-file` specifies the path to the Kubelet’s configuration file.

Default: `/host-var/lib/kubelet/config.yaml`

**Example**

```bash
$ nfd-topology-updater -kubelet-config-file=/var/lib/kubelet/config.yaml
```

The `-no-publish` flag disables all communication with the nfd-master, making it a dry run flag for nfd-topology-updater. NFD Topology Updater runs resource hardware topology detection normally, but no CR requests are sent to nfd-master.
Default: \texttt{false}

Example

\texttt{\$ nfd-topology-updater -no-publish}

4.5.2.1. \texttt{-oneshot}

The \texttt{-oneshot} flag causes the NFD Topology Updater to exit after one pass of resource hardware topology detection.

Default: \texttt{false}

Example

\texttt{\$ nfd-topology-updater -oneshot -no-publish}

\texttt{-podresources-socket}

The \texttt{-podresources-socket} flag specifies the path to the Unix socket where kubelet exports a gRPC service to enable discovery of in-use CPUs and devices, and to provide metadata for them.

Default: \texttt{/host-var/lib/lib/kubelet/pod-resources/kubelet.sock}

Example

\texttt{\$ nfd-topology-updater -podresources-socket=/var/lib/kubelet/pod-resources/kubelet.sock}

\texttt{-server}

The \texttt{-server} flag specifies the address of the nfd-master endpoint to connect to.

Default: \texttt{localhost:8080}

Example

\texttt{\$ nfd-topology-updater -server=nfd-master.nfd.svc.cluster.local:443}

\texttt{-server-name-override}

The \texttt{-server-name-override} flag specifies the common name (CN) which to expect from the nfd-master TLS certificate. This flag is mostly intended for development and debugging purposes.

Default: empty

Example

\texttt{\$ nfd-topology-updater -server-name-override=localhost}

\texttt{-sleep-interval}

The \texttt{-sleep-interval} flag specifies the interval between resource hardware topology re-examination and custom resource updates. A non-positive value implies infinite sleep interval and no re-detection is done.

Default: \texttt{60s}
-watch-namespace
The -watch-namespace flag specifies the namespace to ensure that resource hardware topology examination only happens for the pods running in the specified namespace. Pods that are not running in the specified namespace are not considered during resource accounting. This is particularly useful for testing and debugging purposes. A * value means that all of the pods across all namespaces are considered during the accounting process.

Default: *

Example

```
$ nfd-topology-updater -watch-namespace=rte
```