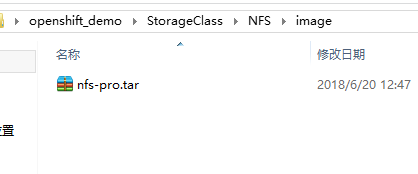
OpenShift Demo Deploy

# StorageClass NFS 部署

## 使用镜像：



registry.example.com:5000/nfs-client-provisioner:latest

## 部署：

``` bash

oc create -f rbac.yaml

oc adm policy add-scc-to-user hostmount-anyuid system:serviceaccount:default:nfs-client-provisioner

oc adm policy add-cluster-role-to-user nfs-client-provisioner-runner system:serviceaccount:default:nfs-client-provisioner

oc create -f deployer.yaml

oc create -f storageclass.yaml

```

### rbac.yaml

kind: ClusterRole

apiVersion: v1

metadata:

name: nfs-client-provisioner-runner

rules:

- apiGroups: [""]

resources: ["persistentvolumes"]

verbs: ["get", "list", "watch", "create", "delete"]

- apiGroups: [""]

resources: ["persistentvolumeclaims"]

verbs: ["get", "list", "watch", "update"]

- apiGroups: ["storage.k8s.io"]

resources: ["storageclasses"]

verbs: ["get", "list", "watch"]

- apiGroups: [""]

resources: ["events"]

verbs: ["list", "watch", "create", "update", "patch"]

---

apiVersion: v1

kind: ServiceAccount

metadata:

name: nfs-client-provisioner

### deployer.yaml

kind: Deployment

apiVersion: extensions/v1beta1

metadata:

name: nfs-client-provisioner

spec:

replicas: 1

strategy:

type: Recreate

template:

metadata:

labels:

app: nfs-client-provisioner

spec:

serviceAccountName: nfs-client-provisioner

containers:

- name: nfs-client-provisioner

image: registry.example.com:5000/nfs-client-provisioner:latest

volumeMounts:

- name: nfs-client-root

mountPath: /persistentvolumes

env:

- name: PROVISIONER\_NAME

value: example.com/nfs

- name: NFS\_SERVER

value: lb.example.com

- name: NFS\_PATH

value: /srv/nfs/storageclass

volumes:

- name: nfs-client-root

nfs:

server: lb.example.com

path: /srv/nfs/storageclass

### storageclass.yaml

kind: StorageClass

apiVersion: storage.k8s.io/v1beta1

metadata:

name: example-nfs

annotations:

storageclass.beta.kubernetes.io/is-default-class: "true"

labels:

kubernetes.io/cluster-service: "true"

provisioner: example.com/nfs

## 使用：

### test\_pvc.yaml

kind: PersistentVolumeClaim

apiVersion: v1

metadata:

name: nfs-5m

annotations:

volume.beta.kubernetes.io/storage-class: "example-nfs"

spec:

accessModes:

- ReadWriteMany

resources:

requests:

storage: 5Mi

# Nexus 服务部署

## 使用镜像：

registry.example.com:5000/nexus3:latest

## 部署：

两种办法：

一 使用oc命令

二 使用template

### Command：

**## Use Image**

\* registry.example.com:5000/nexus3:latest

**## deployer nexus3**

```bash

# create imagestorm

oc create is nexus3

# import image to imagestorm nexus3

oc import-image nexus3:latest --from=registry.example.com:5000/nexus3:latest --insecure=true

# deploy nexus3

oc new-app nexus3

# create route nexus3 from svc nexus3

oc expose svc nexus3

# pause dc nexus3

oc rollout pause dc nexus3

# Config DC

oc patch dc nexus3 --patch='{ "spec": { "strategy": { "type": "Recreate" }}}'

# Set resources

oc set resources dc nexus3 --limits=memory=2Gi --requests=memory=1Gi

# create pvc

echo "apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: nexus-pvc

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 20Gi" | oc create -f -

# Set volume

oc set volume dc/nexus3 --add --overwrite --name=nexus3-volume-1 --mount-path=/nexus-data/ --type persistentVolumeClaim --claim-name=nexus-pvc

# Set probe

oc set probe dc/nexus3 --liveness --failure-threshold 3 --initial-delay-seconds 60 -- echo ok

oc set probe dc/nexus3 --readiness --failure-threshold 3 --initial-delay-seconds 60 --get-url=http://:8081/repository/maven-public/

# Config OK , resume DC

oc rollout resume dc nexus3

```

### Templete：

oc create -f nexus3.yaml

oc new-app --template=nexus3 --param APPLICATION\_NAME=nexus3 \

--param IMAGE\_NAME=..... \

nexus3. yaml

apiVersion: v1

kind: Template

metadata:

name: nexus3

objects:

- apiVersion: v1

kind: DeploymentConfig

metadata:

labels:

app: ${APPLICATION\_NAME}

name: ${APPLICATION\_NAME}

spec:

replicas: 1

selector:

app: ${APPLICATION\_NAME}

deploymentconfig: ${APPLICATION\_NAME}

strategy:

type: Recreate

template:

metadata:

labels:

app: ${APPLICATION\_NAME}

deploymentconfig: ${APPLICATION\_NAME}

spec:

containers:

- image: ${IMAGE\_NAME}

imagePullPolicy: Always

livenessProbe:

exec:

command:

- echo

- ok

failureThreshold: 3

initialDelaySeconds: 60

periodSeconds: 10

successThreshold: 1

timeoutSeconds: 1

name: ${APPLICATION\_NAME}

ports:

- containerPort: 8081

protocol: TCP

readinessProbe:

failureThreshold: 3

httpGet:

path: /repository/maven-public/

port: 8081

scheme: HTTP

initialDelaySeconds: 60

periodSeconds: 10

successThreshold: 1

timeoutSeconds: 1

resources:

limits:

memory: ${M\_L}Gi

requests:

memory: 1Gi

volumeMounts:

- mountPath: /nexus-data

name: ${APPLICATION\_NAME}-volume-1

volumes:

- name: ${APPLICATION\_NAME}-volume-1

persistentVolumeClaim:

claimName: ${APPLICATION\_NAME}-pvc

test: false

# svc

- apiVersion: v1

kind: Service

metadata:

labels:

app: ${APPLICATION\_NAME}

name: ${APPLICATION\_NAME}

spec:

ports:

- name: 8081-tcp

port: 8081

protocol: TCP

targetPort: 8081

selector:

app: ${APPLICATION\_NAME}

deploymentconfig: ${APPLICATION\_NAME}

sessionAffinity: None

type: ClusterIP

# route

- apiVersion: v1

kind: Route

metadata:

labels:

app: ${APPLICATION\_NAME}

name: ${APPLICATION\_NAME}

spec:

port:

targetPort: 8081-tcp

to:

kind: Service

name: ${APPLICATION\_NAME}

weight: 100

wildcardPolicy: None

# pvc

- apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: ${APPLICATION\_NAME}-pvc

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: ${P\_S}Gi

parameters:

- description: The name for the application.

displayName: Application Name

name: APPLICATION\_NAME

required: true

value: nexus3

- description: The application use image

displayName: Image Name

name: IMAGE\_NAME

required: true

value: "registry.example.com:5000/nexus3:latest"

- description: The application memory limits

displayName: Memory limits (Gi)

name: M\_L

required: true

value: "2"

- description: The application PVC Size

displayName: PVC Size (Gi)

name: P\_S

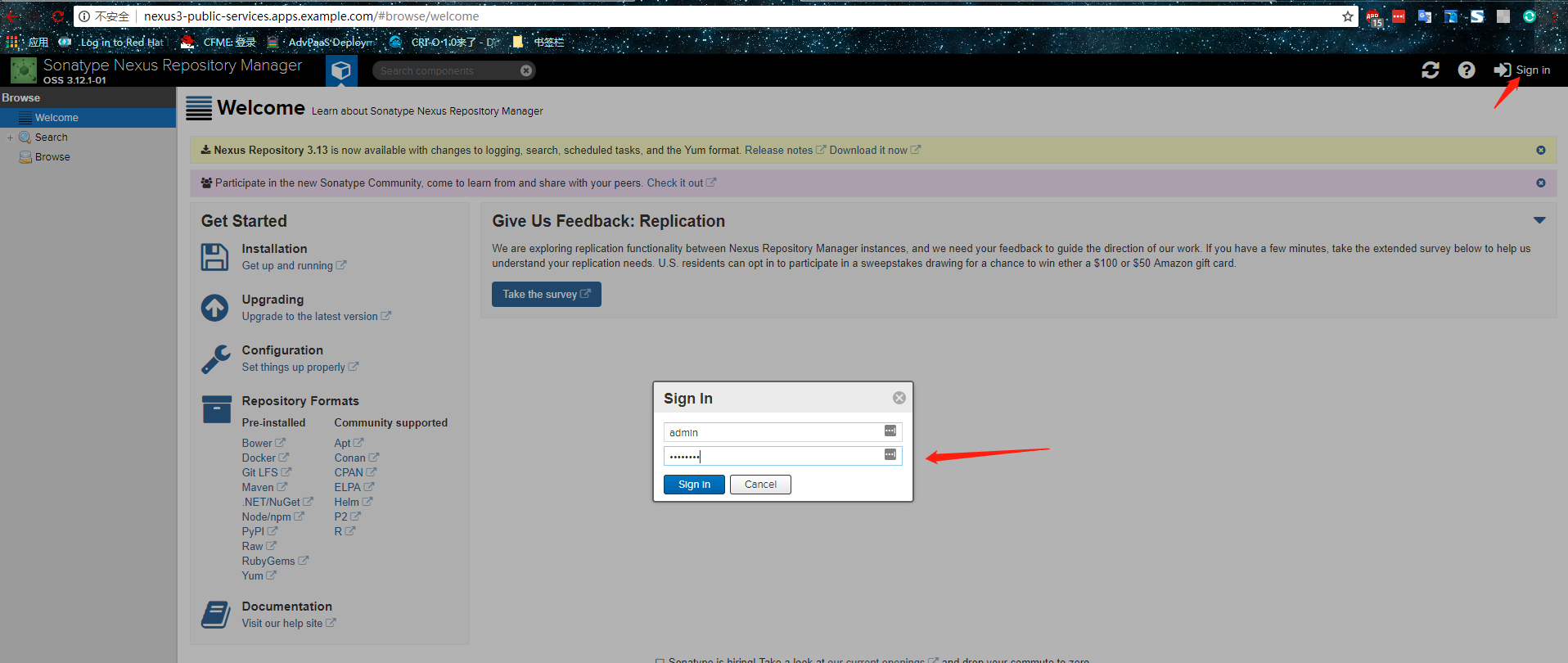
required: true

value: "10"

## 使用：

浏览器打开route , 登陆nexus

admin/admin123



点击 设置图标

选择 Reposltory -- Create reposltory

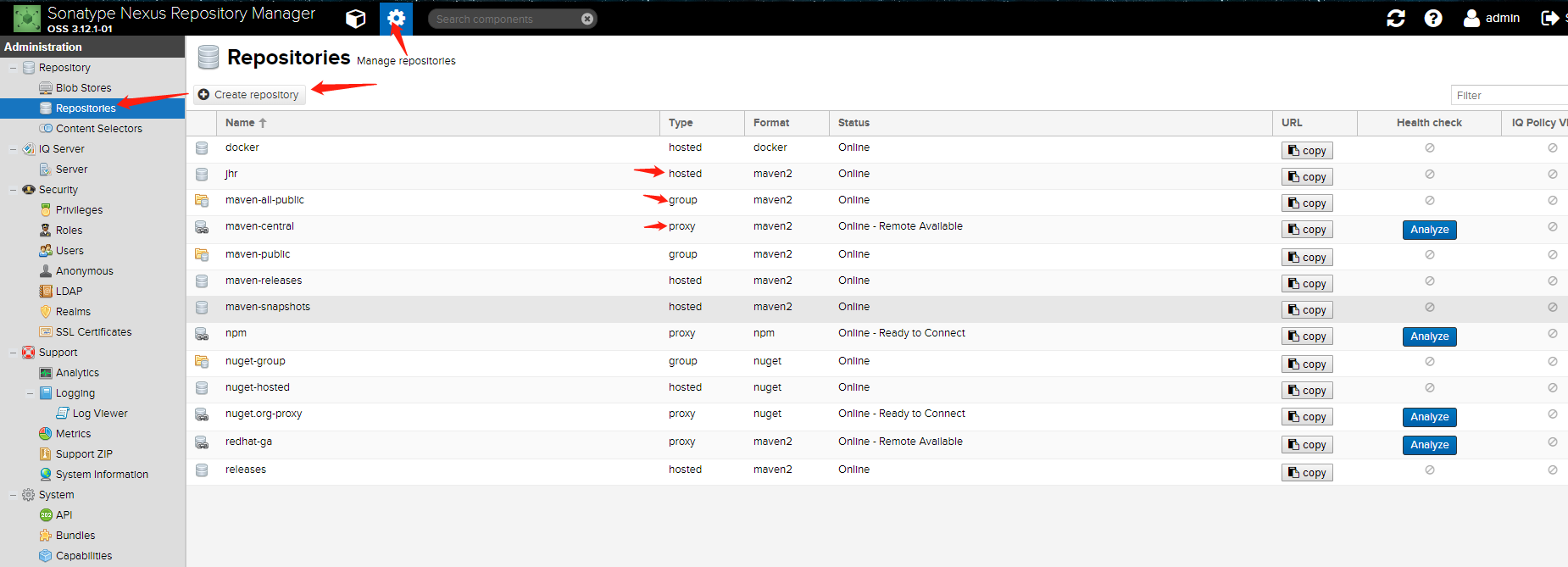
支持 Docker registry / maven /pypi /npm/ yum

repo 分为三种：

group 组 (可将三种放置在一个组中，对外提供组的url)

hosted 本地repo

proxy 代理（可自动代理官方等repo）



# Gogs 服务部署

## 使用镜像：

registry.example.com:5000/wkulhanek/gogs:11.34

## 部署：

oc create is gogs

oc import-image gogs:11.34 --from=registry.example.com:5000/wkulhanek/gogs:11.34 --insecure=true

oc new-app postgresql-persistent --param DATABASE\_SERVICE\_NAME=gogs-db --param POSTGRESQL\_DATABASE=gogs --param POSTGRESQL\_USER=gogs --param POSTGRESQL\_PASSWORD=gogs --param VOLUME\_CAPACITY=4Gi -lapp=postgresql\_gogs

oc new-app gogs:11.34 -lapp=gogs

echo "apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: gogs-data

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 10Gi" | oc create -f -

oc set volume dc/gogs --add --overwrite --name=gogs-volume-1 --mount-path=/data/ --type persistentVolumeClaim --claim-name=gogs-data

oc expose svc gogs

1. In a web browser, navigate to **http://gogsroute** where **gogsroute** is the route you just created.
   * When using the install function of Gogs:
     + The application URL is **http://gogsroute** (**gogsroute** varies based on your environment).
     + The database host points to the PostgreSQL service on port **5432**.
     + The **Run User** parameter is set to **gogs**.
     + All other database parameters match what you specified when creating the PostgreSQL database.
2. Set up Gogs with these values:
   * Database Type: **PostgreSQL**
   * Host: **postgresql:5432**
   * User: **gogs**
   * Password: **gogs**
   * Database Name: **gogs**
   * Run User: **gogs**
   * Application URL: **http://gogsroute**

oc exec $(oc get pod | grep "^gogs" | awk '{print $1}') -- cat /opt/gogs/custom/conf/app.ini >$HOME/app.ini

oc create configmap gogs --from-file=$HOME/app.ini

oc set volume dc/gogs --add --overwrite --name=config-volume -m /opt/gogs/custom/conf/ -t configmap --configmap-name=gogs

## 使用：

oc get route gogs --template='{{ .spec.host }}'

1. Log in to Gogs and create an organization named **CICDLabs**.
2. Under the **CICDLabs** organization, create a repository called **openshift-tasks**.
   * Do not make this a **Private** repository.
3. On your client VM, clone the source code from GitHub and push it to Gogs:

|  |  |
| --- | --- |
|  | Make sure to replace **<gogs\_user>** and **<gogs\_password>** with your credentials. |

1. cd $HOME
2. git clone https://github.com/wkulhanek/openshift-tasks.git
3. cd $HOME/openshift-tasks
4. git remote add gogs http://<gogs\_user>:<gogs\_password>@$(oc get route gogs -n xyz-gogs --template='{{ .spec.host }}')/CICDLabs/openshift-tasks.git
5. git push -u gogs master
6. Set up **nexus\_settings.xml** for local builds, making sure that **<url>** points to your specific Nexus URL:
7. <?xml version="1.0"?>
8. <settings>
9. <mirrors>
10. <mirror>
11. <id>Nexus</id>
12. <name>Nexus Public Mirror</name>
13. <url>http://nexus3-xyz-nexus.apps.$GUID.example.opentlc.com/repository/maven-all-public/</url>
14. <mirrorOf>\*</mirrorOf>
15. </mirror>
16. </mirrors>
17. <servers>
18. <server>
19. <id>nexus</id>
20. <username>admin</username>
21. <password>admin123</password>
22. </server>
23. </servers>

</settings>

|  |  |
| --- | --- |
|  | This file is located in the **$HOME/openshift-tasks** directory. |

1. Commit and push the updated settings files to Gogs:
2. git commit -m "Updated Settings" nexus\_settings.xml nexus\_openshift\_settings.xml

git push gogs master

# SonarQube 部署

Set Up SonarQube

1. Create a new project called **xyz-sonarqube** with a display name of **Shared Sonarqube**, replacing **xyz** with your initials.

oc new-project xyz-sonarqube --display-name "Shared Sonarqube"

1. Deploy a persistent PostgreSQL database.
   * When deploying the template, pick sensible values for the **POSTGRESQL\_USER**, **POSTGRESQL\_PASSWORD**, **POSTGRESQL\_DATABASE**, and **VOLUME\_CAPACITY** parameters.
   * Make sure that your database is fully up before moving to the next step.

oc new-app --template=postgresql-persistent --param POSTGRESQL\_USER=sonar --param POSTGRESQL\_PASSWORD=sonar --param POSTGRESQL\_DATABASE=sonar --param VOLUME\_CAPACITY=4Gi --labels=app=sonarqube\_db

1. Deploy the SonarQube image (**wkulhanek/sonarqube:6.7.4**) available in DockerHub.
   * The image expects **SONARQUBE\_JDBC\_USERNAME**, **SONARQUBE\_JDBC\_PASSWORD**, and **SONARQUBE\_JDBC\_URL** in the environment.
   * The correct setting for **SONARQUBE\_JDBC\_URL** is **SONARQUBE\_JDBC\_URL=jdbc:postgresql://postgresql/<dbname>** where "<dbname>" is the name you picked when you set up PostgreSQL.

|  |  |
| --- | --- |
|  | The source for the Docker image is located at <https://github.com/wkulhanek/docker-openshift-sonarqube.git>. |

1. oc new-app --docker-image=wkulhanek/sonarqube:6.7.4 --env=SONARQUBE\_JDBC\_USERNAME=sonar --env=SONARQUBE\_JDBC\_PASSWORD=sonar --env=SONARQUBE\_JDBC\_URL=jdbc:postgresql://postgresql/sonar --labels=app=sonarqube
2. Pause rollouts for the created SonarQube deployment configuration so you can make a few more changes to the deployment configuration.
3. Create a route for SonarQube.
4. oc rollout pause dc sonarqube

oc expose service sonarqube

1. Create a PVC and mount it at **/opt/sonarqube/data**.
2. echo "apiVersion: v1
3. kind: PersistentVolumeClaim
4. metadata:
5. name: sonarqube-pvc
6. spec:
7. accessModes:
8. - ReadWriteOnce
9. resources:
10. requests:
11. storage: 4Gi" | oc create -f -

oc set volume dc/sonarqube --add --overwrite --name=sonarqube-volume-1 --mount-path=/opt/sonarqube/data/ --type persistentVolumeClaim --claim-name=sonarqube-pvc

1. Set the resources.
   * SonarQube is a heavy application. The following parameters are suggested:
     + Memory request: 1.5Gi
     + Memory limit: 3Gi
     + CPU request: 1 CPU
     + CPU limit: 2 CPUs
2. Set the deployment strategy.
   * Because SonarQube is using **Elasticsearch** under the covers, it needs a **Recreate** deployment strategy rather than the default **Rolling** deployment strategy.
   * oc set resources dc/sonarqube --limits=memory=3Gi,cpu=2 --requests=memory=2Gi,cpu=1

oc patch dc sonarqube --patch='{ "spec": { "strategy": { "type": "Recreate" }}}'

1. To ensure proper operation of your service, add liveness and readiness probes.
2. oc set probe dc/sonarqube --liveness --failure-threshold 3 --initial-delay-seconds 40 -- echo ok

oc set probe dc/sonarqube --readiness --failure-threshold 3 --initial-delay-seconds 20 --get-url=http://:9000/about

1. Finally, resume deployment of the SonarQube deployment configuration to roll out all changes at once.

oc rollout resume dc sonarqube

1. Once SonarQube has fully started, log in via the exposed route. The default user ID is **admin** and password is **admin**.

# Jenkins 部署

1. Create a new project called **xyz-jenkins** with a display name of **Shared Jenkins**.

oc new-project xyz-jenkins --display-name "Shared Jenkins"

1. Set up a persistent Jenkins instance with 2 GB of memory and a persistent volume claim of 4 GB.

oc new-app jenkins-persistent --param ENABLE\_OAUTH=true --param MEMORY\_LIMIT=2Gi --param VOLUME\_CAPACITY=4Gi

1. Edit the Jenkins slave pod configuration to allow the Maven slave pod to consume 2Gi of memory when building a JEE application.

# Jenkins Slave 部署

Work with Custom Jenkins Slave Pod

### 6.1. Create Custom Jenkins Slave Pod

The stock Jenkins Maven slave pod does not have **skopeo** installed. However, you need **skopeo** to be available in order to move your built container images into another registry. This means that you need to build a custom slave pod. You simply extend the existing slave pod and install **skopeo** into that pod. Then you need to make this container image available to OpenShift by pushing it into the OpenShift Container Registry. Because you are building this image yourself you can just use your current Jenkins project (**xyz-jenkins**) as the home for the container image.

Your bastion host already has Docker installed. But because you do not have real certificates in your cluster, your Docker registry is an insecure registry. This means that you need to configure your local Docker daemon to allow connecting to your OpenShift Container Registry.

The route to the OpenShift Container Registry is similar to **docker-registry-default.apps.$GUID.example.opentlc.com**.

Enabling and configuring system services as well as building Docker Containers requires **root** permission. Therefore the following section needs to be run as **root** on your Client VM.

Use **sudo -i** to switch to root.

1. On your client VM, add the OpenShift Container Registry to **/etc/containers/registries.conf**:
2. [...]
3. [registries.insecure]
4. registries = ['docker-registry-default.apps.$GUID.example.opentlc.com']

[...]

1. Enable and start Docker if it is not already running:
2. systemctl enable docker

systemctl start docker

1. In your home directory, create a **jenkins-slave-appdev** subdirectory and change into it:
2. mkdir $HOME/jenkins-slave-appdev

cd $HOME/jenkins-slave-appdev

1. In the **jenkins-slave-appdev** directory, create a Dockerfile.
   * Use **docker.io/openshift/jenkins-slave-maven-centos7:v3.9** as the base image.
     + The classroom clusters do not have a proper subscription, so you cannot build any images based on RHEL—but you can use the upstream CentOS image instead.
   * This base image uses a **1001** user as the user to run the slave pod.
   * You need to install **skopeo** as **root**. Be sure to switch to the **root** user before anything you do in the build process and switch back to **1001** after you are done.
   * Install **skopeo**.
   * Save the Dockerfile.
   * FROM docker.io/openshift/jenkins-slave-maven-centos7:v3.9
   * USER root
   * RUN yum -y install skopeo apb && \
   * yum clean all

USER 1001

1. Build the container.
   * When building the container, make sure to tag it using the route to the Docker registry and the name for your Jenkins project.
     + Since you are pushing the container into the OpenShift Container Registry, you need to pick a project for which you are authorized—the easiest one to pick is your Jenkins project.
     + You also need to use the current version number in your tag.
   * The container name needs to be something like **jenkins-slave-maven-appdev**.
     + You can, of course, use any other name—just make sure you are consistent throughout this lab.
   * Your tag needs to look something like this: **<OCP Container Registry Route>/<Your Jenkins Project>/jenkins-slave-maven-appdev:v3.9**
   * You can also choose to build the image first and tag it after it is successfully built.

docker build . -t docker-registry-default.apps.$GUID.example.opentlc.com/xyz-jenkins/jenkins-slave-maven-appdev:v3.9

### 6.2. Publish Custom Slave Pod to OpenShift Container Registry

You have two choices on how to approach this step.

* Use Docker commands to log in to the OpenShift Container Registry using your OpenShift user ID and associated token as the password and then push the tagged image.
  1. Log in to the OpenShift Container Registry:

docker login -u wkulhane-redhat.com -p $(oc whoami -t) docker-registry-default.apps.$GUID.example.opentlc.com

* 1. Push the image to the registry:

docker push docker-registry-default.apps.$GUID.example.opentlc.com/xyz-jenkins/jenkins-slave-maven-appdev:v3.9

* Use **skopeo** to copy the image from the local Docker daemon storage into the OpenShift Container Registry.
  1. You need to specify **--dest-tls-verify=false** because you are pushing to an insecure registry.
  2. You need to specify **--dest-creds=<user>:<token>** as your push credentials.
  3. Make sure you specify the right kind of storage for both source and target locations.

skopeo copy --dest-tls-verify=false --dest-creds=$(oc whoami):$(oc whoami -t) docker-daemon:docker-registry-default.apps.$GUID.example.opentlc.com/xyz-jenkins/jenkins-slave-maven-appdev:v3.9 docker://docker-registry-default.apps.$GUID.example.opentlc.com/xyz-jenkins/jenkins-slave-maven-appdev:v3.9

### 6.3. Register Custom Slave Pod in Jenkins

When your customized Maven slave pod is available in the OpenShift Container Registry, you need to tell Jenkins where to find it and when to use it.

You can use the existing Maven slave image as a template and copy most of the fields from the existing image.

In Jenkins select **Manage Jenkins**, then click on **Configure System** and finally scroll down to the **Cloud** section. Click **Add Pod Template** and select **Kubernetes Pod Template** to add another pod template to Jenkins.

Make sure you get the following settings right:

* **Labels**: This is the name that you use in your pipeline to specify this image. Suggestion: **maven-appdev**.
* **Docker-Image**: The fully qualified name of your Docker image. Use the OpenShift **internal** service name (and port).
* **Memory limit**: Use **2Gi** for the container memory limit.
  1. From the Jenkins home screen, select **Manage Jenkins → Configure System**.
  2. Select **Cloud → Kubernetes → Add Pod Template → Kubernetes Pod Template**:
     + **Name**: **maven-appdev**
     + **Namespace**: <empty>
     + **Labels**: **maven-appdev**
     + **Usage**: Use this node as much as possible
     + **The name of the pod template to inherit from**: <empty>
     + **Containers**: **Add Container** / **Container Template**
     + **Name**: **jnlp**
     + **Docker image**: **docker-registry.default.svc:5000/xyz-jenkins/jenkins-slave-maven-appdev:v3.9**
     + **Always pull image**: <Unchecked>
     + **Working directory**: **/tmp**
     + **Command to run**: <empty>
     + **Arguments to pass to the command**: **${computer.jnlpmac} ${computer.name}**
     + **Allocate pseudo-TTY**: <Unchecked>
  3. Click **Advanced…** to open the advanced container template settings.
     + **Limit Memory**: **2Gi**
     + Click **Advanced…** at the very bottom of the pod template definition (just above **Delete Template**).
     + **Service Account**: **jenkins**
  4. Click **Save** at the bottom of the screen.

### 6.4. Test Custom Slave Pod

Using a simple pipeline, you can now test that your slave pod is working properly and has **skopeo** installed.

1. Create a new Jenkins job of type **Pipeline** and use this test pipeline:
   * Make sure the label you request matches the label you gave your slave definition.
   * node('maven-appdev') {
   * stage('Test skopeo') {
   * sh("skopeo --version")
   * sh("oc whoami")
   * }

}

1. Run the pipeline (click **Build Now**).
   * Expect the console output to look like this (Click the Build Number, then **Console Output**):
   * Started by user wkulhane-redhat.com
   * [Pipeline] node
   * Running on maven-appdev-jw58m in /tmp/workspace/Slave Test
   * [Pipeline] {
   * [Pipeline] stage
   * [Pipeline] { (Test skopeo)
   * [Pipeline] sh
   * [Slave Test] Running shell script
   * + skopeo --version
   * skopeo version 0.1.28
   * [Pipeline] sh
   * [Slave Test] Running shell script
   * + oc whoami
   * system:serviceaccount:xyz-jenkins:jenkins
   * [Pipeline] }
   * [Pipeline] // stage
   * [Pipeline] }
   * [Pipeline] // node
   * [Pipeline] End of Pipeline

Finished: SUCCESS

# 7. Test Local Workstation Build

In order to verify that all of your build tools are set up correctly, it is a good idea to run a test from your client VM using Nexus and SonarQube from your OpenShift installation.

1. First, make sure you can build the **openshift-tasks** application:
2. cd $HOME/openshift-tasks

mvn clean install -DskipTests=true -s ./nexus\_settings.xml

|  |  |
| --- | --- |
|  | Make sure to double-check the output of the build to verify that your Maven dependencies come from Nexus and not the public Internet repository. |

1. Run the unit tests:

mvn test -s ./nexus\_settings.xml

|  |  |
| --- | --- |
|  | Make sure to double-check the output of the build to verify that your Maven dependencies come from Nexus and not the public Internet repository. |

1. Run the Maven deploy tests (replacing **xyz-nexus** with the name of your Nexus project):
2. mvn -s ./nexus\_settings.xml deploy -DskipTests=true \

-DaltDeploymentRepository=nexus::default::http://$(oc get route nexus3 -n xyz-nexus --template='{{ .spec.host }}')/repository/releases

1. Run the Nexus Docker registry tests (replacing **xyz-nexus** with the name of your Nexus project and **xyz-jenkins** with the name of your Jenkins project):

skopeo copy --dest-tls-verify=false --dest-creds=admin:admin123 docker-daemon:docker-registry-default.apps.na39.openshift.opentlc.com/xyz-jenkins/jenkins-slave-maven-appdev:v3.9 docker://$(oc get route nexus-registry -n xyz-nexus --template='{{ .spec.host }}')/xyz-jenkins/jenkins-slave-maven-appdev:v3.9

1. Run the code analysis tests (replacing **xyz-sonarqube** with the name of your Sonarqube project):

mvn sonar:sonar -s ./nexus\_settings.xml -Dsonar.host.url=http://$(oc get route sonarqube -n xyz-sonarqube --template='{{ .spec.host }}')

* + When the build and tests succeed without error, your environment is ready.

|  |  |
| --- | --- |
|  | Do not delete any of these projects. You use this infrastructure throughout the class. |