CI/CD Tools Lab

In this lab, you create the foundation for the practice of continuous integration and continuous deployment (CI/CD). You set up all of the tools necessary to build a complex pipeline in a later lab. The build process integrates Gogs, Nexus, SonarQube, and S2I builds.

Because you are using a shared environment, make sure to prefix every project/namespace with **xyz-**, where "xyz" are your initials. For example, if the project name is **nexus**, the prefixed string becomes **xyz-nexus**.

**Goals**

* Set up Nexus 3 with persistent storage and configure Nexus to cache Red Hat and other build artifacts.
* Set up SonarQube with persistent storage and with PostgreSQL as the back end.
* Set up Gogs with persistent storage and with PostgreSQL as the back end.
* Set up Jenkins with persistent storage.
* Perform a local workstation build that tests whether all tools are set up and configured correctly.

## 2. Set Up Nexus

* Sonatype provides a Nexus 3 image labeled **sonatype/nexus3:latest** in DockerHub.
* Use the **Recreate** deployment strategy rather than **Rolling** to set up Nexus.
* Nexus requires a large amount of memory. It is suggested that you set the memory request to **1Gi** and the memory limit to **2Gi**.
* The Nexus 3 image defines a **VOLUME** at **/nexus-data**.

Follow these steps to set up Nexus:

1. Create a new project named **xyz-nexus** with display name **Shared Nexus**, replacing **xyz** with your initials.

oc new-project xyz-nexus --display-name "Shared Nexus"

1. Deploy the Nexus container image and create a route to the Nexus service. Because you are making a few changes to the deployment configuration, you may want to pause the automatic deployment upon configuration changes.
2. oc new-app sonatype/nexus3:latest
3. oc expose svc nexus3

oc rollout pause dc nexus3

1. Change the deployment strategy from **Rolling** to **Recreate** and set requests and limits for memory.
2. oc patch dc nexus3 --patch='{ "spec": { "strategy": { "type": "Recreate" }}}'

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

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

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

1. Set up liveness and readiness probes for Nexus.
2. 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/

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

oc rollout resume dc nexus3

### 2.1. Configure Nexus

1. Once Nexus is deployed, set up your Nexus repository using the provided script. Use the Nexus 3 default user ID (**admin**) and password (**admin123**):
2. curl -o setup\_nexus3.sh -s https://raw.githubusercontent.com/wkulhanek/ocp\_advanced\_development\_resources/master/nexus/setup\_nexus3.sh
3. chmod +x setup\_nexus3.sh
4. ./setup\_nexus3.sh admin admin123 http://$(oc get route nexus3 --template='{{ .spec.host }}')

rm setup\_nexus3.sh

* + This script creates:
    - A few Maven proxy repositories to cache Red Hat and JBoss dependencies.
    - A **maven-all-public** group repository that contains the proxy repositories for all of the required artifacts.
    - An NPM proxy repository to cache Node.JS build artifacts.
    - A private Docker registry.
    - A **releases** repository for the WAR files that are produced by your pipeline.
    - A Docker registry in Nexus listening on port 5000. OpenShift does not know about this additional endpoint, so you need to create an additional route that exposes the Nexus Docker registry for your use.

1. Create a Service called **nexus-registry** that exposes port 5000 from the deployment configuration **nexus3**.
2. Create an OpenShift route called **nexus-registry** that uses **edge** termination for the TLS encryption and exposes port 5000.
3. oc expose dc nexus3 --port=5000 --name=nexus-registry

oc create route edge nexus-registry --service=nexus-registry --port=5000

1. Confirm your routes:

oc get routes -n xyz-nexus

**Sample Output**

NAME HOST/PORT PATH SERVICES PORT TERMINATION WILDCARD

nexus-registry nexus-registry-xyz-nexus.apps.$GUID.example.opentlc.com nexus-registry 5000 edge None

nexus3 nexus3-xyz-nexus.apps.$GUID.example.opentlc.com nexus3 8081-tcp None

## 3. 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**.

## 4. Set Up Gogs

Gogs is an open source GitHub clone that can be deployed in a local infrastructure. It requires a PostgreSQL or MySQL database with persistent storage as well as a persistent volume to store its own data.

Gogs is unique in that it must be configured after it is deployed. The database connection as well as other settings must be configured.

Gogs writes the configuration to a file on the local container. Because containers are ephemeral, the Gogs container loses this configuration every time the pod running this Gogs container is redeployed. To prevent this, the configuration file needs to be saved in persistent storage, and a ConfigMap is a good solution for this.

1. Create a new project named **xyz-gogs** with a display name of **Shared Gogs**, replacing **xyz** with your initials.

oc new-project xyz-gogs --display-name "Shared Gogs"

1. Deploy a PostgreSQL database server with persistent storage.
   * There is a **postgresql-persistent** template available in OpenShift.
   * Make sure to add a PostgreSQL user ID, password, and database name when deploying the template.
   * Volume claims of up to 4 GB are supported in the environment.

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

1. Deploy a Gogs server.
   * There is a Docker image for Gogs available at **wkulhanek/gogs:11.34**.

oc new-app wkulhanek/gogs:11.34 -lapp=gogs

1. Add persistent storage for Gogs and attach it to **/data**.
2. echo "apiVersion: v1
3. kind: PersistentVolumeClaim
4. metadata:
5. name: gogs-data
6. spec:
7. accessModes:
8. - ReadWriteOnce
9. resources:
10. requests:
11. storage: 4Gi" | oc create -f -

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

1. Expose the service as a route and retrieve the generated route.
2. oc expose svc gogs

oc get route 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**
3. Click **Install Gogs**.
4. Retrieve the configuration file from the Gogs pod and store it in your **$HOME** directory.
   * The location of the configuration file in the container is **/opt/gogs/custom/conf/app.ini**.

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

1. Create a ConfigMap using the Gogs configuration file.

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

1. Update the Gogs deployment configuration to mount the ConfigMap as a volume at **/opt/gogs/custom/conf**.

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

1. Wait until the redeployment finishes, then navigate back to the Gogs home page (**http://gogsroute**).
2. Register a new user—the first registered user becomes the administrator for Gogs.
3. Log in to Gogs as the user you just registered as.

### 4.1. Install openshift-tasks Source Code into Gogs

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

## 5. Set Up Jenkins

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.

## 6. 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. |