Table of Contents

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.

1. Provision Lab Environment

If you previously set up an environment for this class, you can skip this section and go directly to the Set Up Nexus section.

In this section, you provision the lab environment to provide access to all of the components required to perform the labs, including access to the OPENTLC OpenShift Container Platform portal (https://master.na37.openshift.opentlc.com/). The lab environment is a shared cloud-based environment, so that you can access it over the Internet from anywhere. However, do not expect performance to match a dedicated environment.

1. Go to the OPENTLC lab portal (https://labs.opentlc.com/) and use your OPENTLC credentials to log in.

   > If you do not remember your password, go to the OPENTLC Account Management page (https://www.opentlc.com/account/) to reset your password.

2. Navigate to **Services** → **Catalogs** → **All Services** → **OPENTLC OpenShift Labs**.
3. On the left, select **OPENTLC OpenShift 3.7 Shared Access**.
4. On the right, click **Order**.
5. On the bottom right, click **Submit**.

   > Do not select **App Control** → **Start** after ordering the lab. You are simply requesting access to the shared environment so there is nothing to start.

   - After a few minutes, expect to receive an email with instructions on how to connect to the environment.
6. While you are waiting for the environment to build, read the email carefully and specifically note the following:
   - The email includes a URL for the OpenShift master similar to `https://master.na37.openshift.opentlc.com`.
     - This URL varies based on the region where you are located—in this example, the region is `na37`. Whenever you see "${region}" in a URL from now on, make sure to replace it with the region that your environment was provisioned in.

1.1. Deploy Client

In the labs for this course, you use the command line for some of the tasks. The labs assume that you are using the provided client VM. You may use your own laptop or workstation for this if you are able to install the required software. This is recommended only for experienced users, because it may cause technical issues due to differences between local and virtual environments.

In this section, you deploy the client VM for this course.

1. Go to the OPENTLC lab portal (https://labs.opentlc.com/) and use your OPENTLC credentials to log in.
2. Navigate to **Services** → **Catalogs** → **All Services** → **OPENTLC OpenShift Labs**.
3. On the left, select **OpenShift 3.7 Client VM**.
4. On the right, click **Order**.
5. On the bottom right, click **Submit**.

   > Do not select **App Control** → **Start** after ordering the client VM. This is not necessary in this environment. Selecting **Start** may corrupt the lab environment or cause other complications.

   - After a few minutes, expect to receive an email with instructions on how to connect to the environment.
6. Read the email carefully and specifically note the following:
   - The email includes a hostname for your Client VM.
   - Remember this hostname. It will look something like (where `xxxx` is a unique GUID for your Client VM):

```
ocplab-xxxx-oslab.opentlc.com
```

1.2. Start Client After Shut Down

To conserve resources, the client VM shuts down automatically after eight hours. In this section, you restart the client VM for this course after it has shut down automatically.

1. Go to the OPENTLC lab portal (https://labs.opentlc.com/) and use your OPENTLC credentials to log in.
2. Navigate to **Services** → **My Services** (this should be the screen shown right after logging in).
3. In the list of your services, select your client VM.
4. Select **App Control** → **Start** to start your client VM.
5. Select **Yes** at the **Are you sure?** prompt.
6. On the bottom right, click **Submit**.

After a few minutes, expect to receive an email letting you know that the Client VM has been started.

1.3. Share Public Key with OPENTLC

To access any of your lab systems via SSH, you must use your personal OPENTLC SSO username and public SSH key.

If you have not already done so, you must provide a public SSH key to the OPENTLC authentication system:

1. Go to the OPENTLC Account Management page (https://www.opentlc.com/account/).
2. Click **Update SSH Key** and log in using your OPENTLC credentials.
3. Paste your public key in that location.

For more information on generating SSH keys, see Setting Up an SSH Key Pair (https://www.opentlc.com/ssh.html).

1.4. Test Server Connections

The `ocplab` host—also called the client VM—serves as an access point into the environment and is not part of the OpenShift environment.

1. Connect to your client VM:

```
ssh -i ~/.ssh/yourprivatekey.key opentlc-
username@ocplab-$GUID.oslab.opentlc.com
```

- Remember to replace `$GUID` with your unique GUID from the welcome e-mail.
- Red Hat recommends that you create an environment variable so that you do not have to type it each time.

Example

```
Laptop$ export GUID=c3po
Laptop$ ssh -i ~/.ssh/mykey.key wkulhane-
redhat.com@ocplab-$GUID.oslab.opentlc.com
(root password is: r3dh4t1!)
```

1.5. Connect to OpenShift Cluster

Once you are connected to your client VM, you can log in to the OpenShift cluster.

1. Use the `oc login` command to log in to the cluster, making sure to replace the URL with your cluster's URL:

```
oc login -u <Your OPENTLC UserID> -p <Your OPENTLC Password>
https://master.na37.openshift.opentlc.com
```

If you see a warning about an insecure certificate, type `y` to connect insecurely.

Sample Output

```
Login successful.

You don't have any projects. You can try to create a new project, by running

    oc new-project <projectname>
```

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.
2. 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.
3. Change the deployment strategy from `Rolling` to `Recreate` and set requests and limits for memory.
4. Create a persistent volume claim (PVC) and mount it at `/nexus-data`.
5. Set up liveness and readiness probes for Nexus.
6. Finally, resume deployment of the Nexus deployment configuration to roll out all changes at once.

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`):

```
curl -o setup_nexus3.sh -s
https://raw.githubusercontent.com/wkulhanek/ocp_advanced_development_resources/
master/nexus/setup_nexus3.sh
chmod +x setup_nexus3.sh
./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.

2. Create an OpenShift route called `nexus-registry` that uses `edge` termination for the TLS encryption and exposes port 5000.
3. 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.na37.openshift.opentlc.com
nexus-registry    5000       edge          None
nexus3            nexus3-xyz-nexus.apps.na37.openshift.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.
2. Deploy a persistent PostgreSQL database.
   - When deploying the template, pick sensible values for the `POSTGRESQL_USER`, `POSTGRESQL_PASSWORD`, `POSTGRESQL_DATABASE`, and `VOLUME_CAPACITY` parameters.
   - Deploy the PostgreSQL template using `oc new-app`, not the OpenShift web console.
   - Make sure that your database is fully up before moving to the next step.
3. Deploy the SonarQube image (`wkulhanek/sonarqube:6.7.2`) 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 (https://github.com/wkulhanek/docker-openshift-sonarqube.git).

4. Pause rollouts for the created SonarQube deployment configuration so you can make a few more changes to the deployment configuration.
5. Create a route for SonarQube.

6. Create a PVC and mount it at `/opt/sonarqube/data`.
7. 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
8. 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.
9. To ensure proper operation of your service, add liveness and readiness probes.
10. Finally, resume deployment of the SonarQube deployment configuration to roll out all changes at once.
11. 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.
2. Deploy a PostgreSQL database server with persistent storage.
    - There is a `postgresql-persistent` template available in OpenShift. Make sure to deploy this template using `oc new-app`, not the OpenShift web console.
    - 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.
3. Deploy a Gogs server.
    - There is a Docker image for Gogs available at `wkulhanek/gogs:11.34`.
4. Add persistent storage for Gogs and attach it to `/data`.
5. Expose the service as a route and retrieve the generated route.
6. 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.
7. 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`.

8. Create a ConfigMap using the Gogs configuration file.
9. Update the Gogs deployment configuration to mount the ConfigMap as a volume at `/opt/gogs/custom/conf`.
10. Wait until the redeployment finishes, then navigate back to the Gogs home page (`http://gogsroute`).
11. Register a new user—the first registered user becomes the administrator for Gogs.
12. 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.

```
cd $HOME
git clone https://github.com/wkulhanek/openshift-tasks.git
cd $HOME/openshift-tasks
git remote add gogs http://<gogs_user>:<gogs_password>@$(oc get route gogs -n
xyz-gogs --template='{{ .spec.host }}')/CICDLabs/openshift-tasks.git
git push -u gogs master
```

4. Set up `nexus_settings.xml` for local builds, making sure that `<url>` points to your specific Nexus URL:

```
<?xml version="1.0"?>
<settings>
  <mirrors>
    <mirror>
      <id>Nexus</id>
      <name>Nexus Public Mirror</name>
      <url>http://nexus3-xyz-
nexus.apps.na37.openshift.opentlc.com/repository/maven-all-public/</url>
      <mirrorOf>*</mirrorOf>
    </mirror>
  </mirrors>
  <servers>
  <server>
    <id>nexus</id>
    <username>admin</username>
    <password>admin123</password>
  </server>
</servers>
</settings>
```

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

5. Commit and push the updated settings files to Gogs:

```
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`.
2. Set up a persistent Jenkins instance with 2 GB of memory and a persistent volume claim of 4 GB.
3. 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.na37.openshift.opentlc.com`.

1. On your client VM, add the OpenShift Container Registry to `/etc/containers/registry.conf`:

```
[...]
[registries.insecure]
registries = ['docker-registry-default.apps.na37.openshift.opentlc.com']
[...]
```

2. Enable and start Docker if it is not already running:

```
systemctl enable docker
systemctl start docker
```

3. In your home directory, create a `jenkins-slave-appdev` subdirectory and change into it:

```
mkdir $HOME/jenkins-slave-appdev
cd  $HOME/jenkins-slave-appdev
```

4. In the `jenkins-slave-appdev` directory, create a Dockerfile.
   - Use `docker.io/openshift/jenkins-slave-maven-centos7:v3.7` 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.
5. 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:v3.7`
   - You can also choose to build the image first and tag it after it is successfully built.

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

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.

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.

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")
    }
}
```

2. Run the pipeline.
   - Expect the console output to look like this:

```
Started by user wkulhane-redhat.com
[Pipeline] node
Still waiting to schedule task
All nodes of label ‘maven-appdev’ are offline
Running on maven-appdev-29t4r in /tmp/workspace/test
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Test skopeo)
[Pipeline] sh
[test] Running shell script
+ skopeo --version
skopeo version 0.1.26
[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:

```
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.

2. 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.

3. Run the code analysis tests:

```
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.