Table of Contents

Pipelines Lab

In this lab, you explore the practice of continuous integration and continuous deployment (CI/CD). You build a pipeline that delivers the `openshift-tasks` application into a production environment. The build process integrates Gogs, Nexus, SonarQube, and S2I builds. The final deployment into production uses a blue-green deployment. You configure the pipeline to be triggered when a new version of the application is pushed to Gogs. Finally, you integrate the pipeline with the OpenShift web console.

Goals

- Set up OpenShift development, test, and production projects
- Set up a pipeline project in Jenkins
- Check out the source code from Gogs
- Build the application from source using Nexus as a Maven proxy
- Add a tag `x.x-Buildnumber` to the built image
- Run unit tests
- Run code coverage tests
- Add an `x.x` tag to the image
- Save the container image to the Nexus Docker registry
- Deploy the application to production using a blue-green deployment
- Stop the pipeline for approval
- After approval, switch the route over to the new deployment

Prerequisites

- Completion of the *CI/CD Tools* lab
- The following tools available and configured (from the *CI/CD Tools* lab):
  - Jenkins
    - Custom Jenkins Maven slave image with `skopeo` installed
  - Gogs
    - `openshift-tasks-private` repository populated
  - Nexus
    - Configured as a Maven proxy repository
    - Configured as a release repository
    - Configured as a Docker registry
  - SonarQube

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 Projects in OpenShift 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 Projects in OpenShift

In this section, you set up the development and production projects in OpenShift.

1. Create the `xyz-tasks-dev` OpenShift project (replacing `xyz` with your initials) to hold the development version of the `openshift-tasks` application:
   a. Set up the permissions for Jenkins to be able to manipulate objects in the `xyz-tasks-dev` project.
   b. Create a binary build using the `jboss-eap70-openshift:1.6` image stream.
   c. Create a new deployment configuration pointing to `tasks:0.0-0`.
   d. Turn off automatic building and deployment.
   e. Expose the deployment configuration as a service (on port `8080`) and the service as a route.
   f. Create a placeholder ConfigMap (to be updated by the pipeline).
   g. Attach the ConfigMap to the deployment configuration.

```
# Set up Dev Project
oc new-project xyz-tasks-dev --display-name "Tasks Development"
oc policy add-role-to-user edit system:serviceaccount:xyz-
jenkins:jenkins -n xyz-tasks-dev
oc policy add-role-to-user edit system:serviceaccount:xyz-
jenkins:default -n xyz-tasks-dev

# Set up Dev Application
oc new-build --binary=true --name="tasks" jboss-eap70-
openshift:1.6 -n xyz-tasks-dev
oc new-app xyz-tasks-dev/tasks:0.0-0 --name=tasks --allow-
missing-imagestream-tags=true -n xyz-tasks-dev
oc set triggers dc/tasks --remove-all -n xyz-tasks-dev
oc expose dc tasks --port 8080 -n xyz-tasks-dev
oc expose svc tasks -n xyz-tasks-dev
oc create configmap tasks-config --from-literal="application-
users.properties=Placeholder" --from-literal="application-
roles.properties=Placeholder" -n xyz-tasks-dev
oc set volume dc/tasks --add --name=jboss-config --mount-
path=/opt/eap/standalone/configuration/application-
users.properties --sub-path=application-users.properties --
configmap-name=tasks-config -n xyz-tasks-dev
oc set volume dc/tasks --add --name=jboss-config1 --mount-
path=/opt/eap/standalone/configuration/application-
roles.properties --sub-path=application-roles.properties --
configmap-name=tasks-config -n xyz-tasks-dev
```

2. Create the **`xyz-tasks-prod`** OpenShift project (replacing **`xyz`** with your initials) to hold the production versions of the **`openshift-tasks`** application:
   a. Set up the permissions for Jenkins to be able to manipulate objects in the **`xyz-tasks-prod`** project.
   b. Create two new deployment configurations: **`tasks-green`** and **`tasks-blue`** and point both to **`tasks:0.0`**.
   c. Turn off automatic building and deployment for both deployment configurations.
   d. Add placeholder ConfigMaps.
   e. Associate the ConfigMaps with deployment configurations.
   f. Expose the deployment configurations as a service (on port **`8080`**) and the blue service as a route.

```
# Set up Production Project
oc new-project xyz-tasks-prod --display-name "Tasks Production"
oc policy add-role-to-group system:image-puller
system:serviceaccounts:xyz-tasks-prod -n xyz-tasks-dev
oc policy add-role-to-user edit system:serviceaccount:xyz-
jenkins:jenkins -n xyz-tasks-prod
oc policy add-role-to-user edit system:serviceaccount:xyz-
jenkins:default -n xyz-tasks-prod

# Create Blue Application
oc new-app xyz-tasks-dev/tasks:0.0 --name=tasks-blue --allow-
missing-imagestream-tags=true -n xyz-tasks-prod
oc set triggers dc/tasks-blue --remove-all -n xyz-tasks-prod
oc expose dc tasks-blue --port 8080 -n xyz-tasks-prod
oc create configmap tasks-blue-config --from-
literal="application-users.properties=Placeholder" --from-
literal="application-roles.properties=Placeholder" -n xyz-
tasks-prod
oc set volume dc/tasks-blue --add --name=jboss-config --mount-
path=/opt/eap/standalone/configuration/application-
users.properties --sub-path=application-users.properties --
configmap-name=tasks-blue-config -n xyz-tasks-prod
oc set volume dc/tasks-blue --add --name=jboss-config1 --mount-
path=/opt/eap/standalone/configuration/application-
roles.properties --sub-path=application-roles.properties --
configmap-name=tasks-blue-config -n xyz-tasks-prod

# Create Green Application
oc new-app xyz-tasks-dev/tasks:0.0 --name=tasks-green --allow-
missing-imagestream-tags=true -n xyz-tasks-prod
oc set triggers dc/tasks-green --remove-all -n xyz-tasks-prod
oc expose dc tasks-green --port 8080 -n xyz-tasks-prod
oc create configmap tasks-green-config --from-
literal="application-users.properties=Placeholder" --from-
literal="application-roles.properties=Placeholder" -n xyz-
tasks-prod
oc set volume dc/tasks-green --add --name=jboss-config --mount-
path=/opt/eap/standalone/configuration/application-
users.properties --sub-path=application-users.properties --
configmap-name=tasks-green-config -n xyz-tasks-prod
oc set volume dc/tasks-green --add --name=jboss-config1 --
mount-path=/opt/eap/standalone/configuration/application-
```

```
roles.properties --sub-path=application-roles.properties --
configmap-name=tasks-green-config -n xyz-tasks-prod

# Expose Blue service as route to make blue application active
oc expose svc/tasks-blue --name tasks -n xyz-tasks-prod
```

---

3. Build Pipeline

3.1. Create Pipeline Job

1. In Jenkins, create a **New Item**:
   - **Type**: `Pipeline`
   - **Name**: `Tasks`
   - **Pipeline Script**: See below

```
#!groovy

// Run this pipeline on the custom Maven Slave ('maven-appdev')
// Maven Slaves have JDK and Maven already installed
// 'maven-appdev' has skopeo installed as well.
node('maven-appdev') {
  // Define Maven Command. Make sure it points to the correct
  // settings for our Nexus installation (use the service to
  // bypass the router). The file nexus_openshift_settings.xml
  // needs to be in the Source Code repository.
  def mvnCmd = "mvn -s ./nexus_openshift_settings.xml"

  // Checkout Source Code
  stage('Checkout Source') {
    // TBD
  }

  // The following variables need to be defined at the top
level
  // and not inside the scope of a stage - otherwise they would
not
  // be accessible from other stages.
  // Extract version and other properties from the pom.xml
  def groupId    = getGroupIdFromPom("pom.xml")
  def artifactId = getArtifactIdFromPom("pom.xml")
  def version    = getVersionFromPom("pom.xml")

  // Set the tag for the development image: version + build
number
  def devTag  = "0.0-0"
  // Set the tag for the production image: version
  def prodTag = "0.0"

  // Using Maven build the war file
  // Do not run tests in this step
  stage('Build war') {
    echo "Building version ${version}"
    // TBD
  }

  // Using Maven run the unit tests
  stage('Unit Tests') {
```

```
    echo "Running Unit Tests"
    // TBD
  }

  // Using Maven call SonarQube for Code Analysis
  stage('Code Analysis') {
    echo "Running Code Analysis"
    // TBD
  }

  // Publish the built war file to Nexus
  stage('Publish to Nexus') {
    echo "Publish to Nexus"
    // TBD
  }

  // Build the OpenShift Image in OpenShift and tag it.
  stage('Build and Tag OpenShift Image') {
    echo "Building OpenShift container image tasks:${devTag}"
   // TBD
  }

  // Deploy the built image to the Development Environment.
  stage('Deploy to Dev') {
    echo "Deploying container image to Development Project"
    // TBD
  }

  // Run Integration Tests in the Development Environment.
  stage('Integration Tests') {
    echo "Running Integration Tests"
    // TBD
  }

  // Copy Image to Nexus Docker Registry
  stage('Copy Image to Nexus Docker Registry') {
    echo "Copy image to Nexus Docker Registry"
    // TBD
  }

  // Blue/Green Deployment into Production
  // ------------------------------------
  // Do not activate the new version yet.
```

```
    def destApp   = "tasks-green"
    def activeApp = ""

    stage('Blue/Green Production Deployment') {
      // TBD
    }

    stage('Switch over to new Version') {
      // TBD
      echo "Switching Production application to ${destApp}."
      // TBD
    }
  }

  // Convenience Functions to read variables from the pom.xml
  // Do not change anything below this line.
  // --------------------------------------------------------
  def getVersionFromPom(pom) {
    def matcher = readFile(pom) =~ '<version>(.+)</version>'
    matcher ? matcher[0][1] : null
  }
  def getGroupIdFromPom(pom) {
    def matcher = readFile(pom) =~ '<groupId>(.+)</groupId>'
    matcher ? matcher[0][1] : null
  }
  def getArtifactIdFromPom(pom) {
    def matcher = readFile(pom) =~ '<artifactId>(.+)
  </artifactId>'
    matcher ? matcher[0][1] : null
  }
```

3.2. Check Out Source Code

The first step in the pipeline is to get the source code from the **`openshift-tasks-protected`** repository in Gogs.

1. Fill in the correct command(s) in this section:

```
// Checkout Source Code
stage('Checkout Source') {
    // TBD
}
```

2. Consider the following:
   - What is the URL of the source code repository?
     - Do you want to use the service or the route to access Gogs?
   - This is a private repository. How do you specify credentials for Jenkins to authenticate?
3. Confirm that you are ready to run your first pipeline.
   a. Make sure every step completes successfully and that Jenkins can check out your source code.

3.3. Set Image Tags

Now that your source code is checked out, the pipeline can introspect the **`pom.xml`** file to determine the version of the application to be built. It is a best practice to set the development tag to a combination of the version number plus the actual build number from Jenkins. This way, you can run the pipeline multiple times and every time the version number is different. When the build passes all checks and tags the image as ready for production, the tag is just the version number.

1. In your pipeline, set the correct tags in the following section:

```
// Set the tag for the development image: version + build number
def devTag  = "0.0-0"
// Set the tag for the production image: version
def prodTag = "0.0"
```

3.4. Build WAR File

The second step in your pipeline is to build the WAR file from the source code.

1. Fill in the correct command(s) in this section:

```
// Using Maven build the war file
// Do not run tests in this step
stage('Build war') {
  echo "Building version ${devTag}"
  // TBD
}
```

2. Consider the following:
   - The pipeline defines an environment variable, **`mvnCmd`**, as **`mvn -s ./nexus_openshift_settings.xml`**.
   - **`nexus_openshift_settings.xml`** is in the Gogs repository and needs to point to the correct Nexus service.
   - Use **`-DskipTests`** to bypass unit tests. You run unit tests in the next section.

3. Run the pipeline and verify the following:
   a. The build succeeds.
   b. The build pulls build artifacts from Nexus instead of the Internet.
   c. The build does not execute any unit tests.

3.5. Run Unit Tests

Once you build the WAR file, run unit tests.

1. Fill in the correct command(s) in this section:

```
// Using Maven run the unit tests
stage('Unit Tests') {
  echo "Running Unit Tests"
  // TBD
}
```

2. Consider the following:
   - What is the Maven command to run unit tests?
3. Run the pipeline and verify that the unit tests complete successfully.
4. Once the tests complete successfully, you can comment out the step in the pipeline to execute unit tests to save time in subsequent pipeline runs.

3.6. Run Code Coverage Tests Using SonarQube

Once your unit tests succeed, run code coverage tests using SonarQube.

1. Fill in the correct command(s) in this section:

```
// Using SonarQube to run code analysis
stage('Code Analysis') {
  echo "Running Code Analysis"
  // TBD
}
```

2. Consider the following:
   - What is the Maven command to run SonarQube analysis?
   - What is the URL of the SonarQube service?
     - Do you want to use the service or the route?
   - What do you notice in the build log once the analysis is complete?
     - Again, consider if using a service or a route is better in this case.
   - What is the project name in SonarQube?

    - Do you prefer every pipeline run to update the same project in SonarQube or do you prefer individual projects for each pipeline run?
3. Run the pipeline and verify that the code analysis tests complete successfully.
4. Verify the results in SonarQube.
5. Once the tests complete successfully, you can comment out the step to run code coverage tests to save time in subsequent pipeline runs.

3.7. Store WAR File in Nexus

Once your code coverage tests succeed, you can be relatively certain that this current build is OK. The next step is to archive the built WAR file in Nexus.

1. Fill in the correct command(s) in this section:

```
// Publish the built war file to Nexus
stage('Publish to Nexus') {
  echo "Publish to Nexus"
  // TBD
}
```

2. Consider the following:
    - What is the Maven command to archive the WAR file in Nexus?
    - What is the correct URL of the Nexus release repository?
        - Do you want to use the service or the route?
3. Run the pipeline and verify that the archiving completes successfully.
4. Verify the results in Nexus.

3.8. Build Container Image in OpenShift

Once the WAR file has been archived, you can build the container image in OpenShift. Because you already built the WAR file in Jenkins, you can use a binary build to copy the WAR file into the new S2I build.

1. Fill in the correct command(s) in this section:

```
// Build the OpenShift Image in OpenShift and tag it.
stage('Build and Tag OpenShift Image') {
  echo "Building OpenShift container image tasks:${devTag}"
  // TBD
}
```

2. Consider the following:

- Where in your Jenkins workspace is the WAR file that you want to use?
  - What is the filename of the built WAR file?
- Can you use the WAR file from Nexus instead?
  - If so, what is the filename and location of the WAR file in Nexus?
- Make sure you follow the binary build, which blocks the pipeline until the build is finished. Otherwise, you may be tagging the previously built image.
- Your successfully built image has the tag `latest`.
- Which additional tag does the built image need?

3. Run the pipeline and verify that the build completes successfully.
4. Review the build logs in the OpenShift `xyz-tasks-dev` project.
5. Review the image stream in your `xyz-tasks-dev` project to verify that the correct tag was set.

3.9. Deploy Built Image into Development Project

The next step is to deploy the container image that you just built. There is already a `tasks` deployment configuration in the `xyz-tasks-dev` project. It points to the `tasks:0.0-0` image, which does not exist. You need to update the deployment configuration to point to the correct image.

The project also contains a `tasks-config` ConfigMap that is supposed to hold two configuration files for the JBoss EAP image. In a regular S2I build, everything from the `configuration` directory in the source code repository automatically gets copied to the JBoss EAP configuration in the built image. However, because you built the image using a binary build, this did not happen. Therefore, you need to add the configuration files to the deployment configuration using a ConfigMap.

You need to delete the current ConfigMap and create a new one with the two files `./configuration/application-users.properties` and `./configuration/application-roles.properties`. You do not need to worry about associating this ConfigMap with the deployment configuration—you already did that when you set up the project.

Once the ConfigMap is created, you can deploy the application using the updated image and ConfigMap, and then verify that the deployment succeeds.

Finally, you need to check that the service for the deployment configuration reacts to traffic.

1. Fill in the correct command(s) in this section:

```
// Deploy the built image to the Development Environment.
stage('Deploy to Dev') {
  echo "Deploying container image to Development Project"
  // TBD
}
```

2. Consider the following:
   - How do you tell the deployment configuration which image to use?
   - How do you delete an existing ConfigMap?
   - How do you create a new ConfigMap with content from files?
   - How do you trigger the deployment once the deployment configuration points to the correct image?
     - Is there a built-in Jenkins pipeline command you can use?
   - How do you verify that the deployment was successful?
     - Is there a built-in Jenkins pipeline command you can use?
   - How do you verify that the service for the deployment configuration is working?
     - Is there a built-in Jenkins pipeline command you can use?
3. Run the pipeline and verify that the deployment completes successfully.
4. Review the deployment configuration to double-check that the correct image was deployed.
   - You need to compare the image `sha256` hash code with the tag in the Image Stream.
5. Review the ConfigMap to verify that it has the correct data.
6. In a web browser, navigate to your application's route to validate that it is working properly.

3.10. Run Integration Tests

It is a good idea to test the running application/service to ensure that it is still behaving as expected. This is where integration tests come into play.

The `openshift-tasks` application is a simple task management application with a REST interface to create, retrieve, and delete tasks. The `README` file for the application has information about user IDs and possible commands to manipulate tasks.

In this pipeline, you create a task, retrieve it, and then delete that task. This leaves the database in the same state it was in before running the tests. When all commands succeed, you can be confident that your application is working properly.

1. Wait 15 seconds to ensure that the application has started successfully.
2. Create a task:

   Sample Output

```
HTTP/1.1 201 Created
Expires: 0
Cache-Control: no-cache, no-store, must-revalidate
X-Powered-By: Undertow/1
Server: JBoss-EAP/7
Pragma: no-cache
Location: http://tasks.xyz-tasks-
dev.svc.cluster.local:8080/ws/tasks/1
Date: Tue, 20 Feb 2018 16:40:52 GMT
Connection: keep-alive
Content-Length: 0
```

- Note the `201 created` response status.

3. Retrieve the task:

   Sample Output

```
HTTP/1.1 200 OK
Expires: 0
Cache-Control: no-cache, no-store, must-revalidate
X-Powered-By: Undertow/1
Server: JBoss-EAP/7
Pragma: no-cache
Date: Tue, 20 Feb 2018 16:40:52 GMT
Connection: keep-alive
Content-Type: application/xml
Content-Length: 151

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<collection><task id="1" ownerName="tasks">
<title>integration_test_1</title></task></collection>
```

- Note the `200 OK` response status and the content of your task as XML payload.

4. Delete the task:

   Sample Output

```
HTTP/1.1 204 No Content
Expires: 0
Cache-Control: no-cache, no-store, must-revalidate
X-Powered-By: Undertow/1
Server: JBoss-EAP/7
Pragma: no-cache
Date: Tue, 20 Feb 2018 16:40:53 GMT
```

- Note the **`204 No Content`** response status.

5. Fill in the correct command(s) in this section:

```
// Run Integration Tests in the Development Environment.
stage('Integration Tests') {
  echo "Running Integration Tests"
  sleep 15
  // TBD
}
```

6. Consider the following:
   - Which user ID and password combination do you want to use to access the **`openshift-tasks`** application?
   - What is the URL of the task application?
     - Do you want to use the route or the service?
   - What is the best way to call a REST service from within a Jenkins slave pod?
7. Run the pipeline and verify that the integration tests complete as expected.

3.11. Copy Container Image to Nexus Container Registry

Oftentimes the pipeline needs to move the tested container image to another container registry, or another OpenShift cluster entirely, for deployment to production.

In your environment, Nexus is configured to act as a Docker registry. You use **`skopeo`** to copy the tested image from the integrated OpenShift Container Registry to the Nexus container registry. Once the image is copied to Nexus, you need to tag the image as ready for production.

1. Fill in the correct command(s) in this section:

```
// Copy Image to Nexus Docker Registry
stage('Copy Image to Nexus Docker Registry') {
  echo "Copy image to Nexus Docker Registry"
  // TBD
}
```

2. Consider the following:
   - Using the customer `maven-appdev` slave builder image, you can use `skopeo` to manipulate container images.
   - What is the correct command for `skopeo` to copy an image from one registry to another?
   - What is the correct service/route for the OpenShift Container Registry?
   - What is the correct service/route for the Nexus container registry?
   - Both registries need credentials.
     - Which credentials do you use for the OpenShift Container Registry?
     - Which user does the `maven-appdev` pod run under?
     - When specifying a token, do you need to specify the user name?
     - Which credentials do you use for the Nexus container registry?
   - Both registries use self-signed certificates—how do you bypass certificate validation?
   - How can you test your `skopeo` command without running your pipeline repeatedly?
   - What is the correct tag for the image to roll it out into production?
     - What is the source tag for the production tag—is it `'latest'` or something else?
     - Which project are you using to manipulate the Image Stream?
3. Run the pipeline and verify that the container image gets copied from the OpenShift Container Registry to the Nexus container registry successfully.
4. Review the Nexus container registry to validate that the correct image was copied.
   - You need to compare the image `sha256` hash code with the tag in the Image Stream.

3.12. Deploy to Production Using Blue-Green Deployment

Once your container image is safely stored in the Nexus container registry, you can deploy the image into your production environment.

> Because the image is in a registry external to OpenShift, you can deploy the image into a completely different cluster for production. In that case, you need to either enable your cluster to pull from this external registry—or use `skopeo` again to copy the image to the container registry in your production cluster.

In this lab, you are deploying into the production project on the same cluster.

You are using a blue-green deployment methodology to execute the deployment. This means that:

- You need to determine which version of the application (blue or green) is currently active.
- You need to update the correct ConfigMap with the JBoss EAP server configuration as well.
- You deploy the application to the currently inactive application.
- In this pipeline, you need to stop for approval when the new version of the application has been deployed before you switch the route.

1. Fill in the correct command(s) in this section:

```
// Blue/Green Deployment into Production
// -------------------------------------
// Do not activate the new version yet.
def destApp   = "tasks-green"
def activeApp = ""

stage('Blue/Green Production Deployment') {
  // TBD
}
```

2. Consider the following:
   - How do you determine which version of the application is currently active?
     - Can you examine where the route is pointing to?
   - Once you know which deployment configuration to update, how do you update the image that is to be deployed?
   - What is the exact image that is to be deployed?
   - Where is that image located? In which project?
   - Which ConfigMap needs to be updated with the JBoss EAP server configuration?
   - How do you trigger a redeployment once you have updated your image?
   - How do you verify that the deployment was successful?
3. Run the pipeline and verify that the correct deployment configuration was updated.
4. Verify that the deployment configuration points to the correct image (you may need to compare with the `sha256` hash codes to verify this).
5. Verify that your route still points to the currently active configuration and not the one you just deployed.

3.13. Switch Production Application

Your application is now ready for switchover. The application is fully running and ready to receive traffic. You can run another test using the live application in the production environment (cluster) before switching over traffic, if you want. In this lab, you are stopping the pipeline for approval. Once

approved, the pipeline switches the route from the old application to the new one.

1. Fill in the correct command(s) in this section:

```
stage('Switch over to new Version') {
  // TBD
}
```

2. Consider the following:
   - How do you stop the pipeline and ask for approval?
   - How to you change the route to point to the other service?
3. Run the pipeline and double-check that the route changes to the new version of the application once you approve the switch.

3.14. Move Pipeline to Source Code Repository

Now that you are happy with your pipeline, it is a good idea to move the pipeline into a source code repository.

1. In your **`openshift-tasks-private`** directory, create a **`Jenkinsfile`** file and copy the pipeline script from your Jenkins tasks into this file.
2. Add the **`Jenkinsfile`** file to the repository, then commit and push it into the repository.
3. Change the pipeline job definition in Jenkins to point to this **`pipeline as code`** from the Gogs repository.
4. Consider the following:
   - You may need to change the way your pipeline checks out the source code from Gogs.
   - Your Jenkins job may need credentials to retrieve the **`Jenkinsfile`** from Gogs.
5. Make sure to run the pipeline one last time with all stages that you previously commented out enabled.
   - Your pipeline is now complete.

---

4. Set Up Web Deployment Hook

To automate the build whenever new content is pushed to the **`openshift-tasks-private`** repository, set up a Git hook in Gogs. This way, a Jenkins build is triggered whenever code is pushed to the **`openshift-tasks-private`** source repository on Gogs.

1. Find the authorization token in Jenkins:
   a. Open a browser and log in to Jenkins.
   b. In the top right corner, click the down arrow next to your username and select **Configure**.
   c. Click **Show API Token** and make note of the **User ID** and **API Token** that are displayed.

2. Create the web hook in Gogs:
   a. Open a browser, navigate to the Gogs server, log in, and go to the **`CICDLabs/openshift-tasks-private`** repository.
   b. Click **Settings**, and then click **Git Hooks**.
   c. Click the pencil icon next to **post-receive**.
   d. Copy and paste this script into the **Hook Content** field, replacing **`<userid>`** and **`<apiToken>`** with your Jenkins user ID and API token, **`<jenkinsService>`** with the name of your Jenkins service, and **`<jenkinsProject>`** with the name of the OpenShift project your Jenkins service is located in:

```
#!/bin/bash

while read oldrev newrev refname
do
  branch=$(git rev-parse --symbolic --abbrev-ref $refname)
  if [[ "$branch" == "master" ]]; then
    curl -k -X POST --user <userid>:<apiToken>
http://<jenkinsService>.
<jenkinsProject>.svc.cluster.local/job/Tasks/build
  fi
done
```

   - This script signals the Jenkins **`Tasks`** build job each time a commit is made to the master branch.
3. Click **Update Hook**.

---

5. Trigger New Build

Committing a new version of the application source code triggers a new build as a result of the Git hook you configured. It is a good practice to increment the version number each time you make changes to your application. You can increment the version number manually or automatically.

1. Increment the version number:
   a. Change the **`openshift-tasks`** source code to make sure you are seeing an updated version of your application after the pipeline completes.
      - Look near line 45 of the code—where the title of the page is rendered—in the **`$HOME/openshift-tasks/src/main/webapp/index.js`** file.
   b. Increment the version number of the project each time code is pushed using Maven—for example, updating the version number (**`VERSION=1.0`**) with the next minor or major number (**`VERSION=1.1`**):

```
cd $HOME/openshift-tasks
export VERSION=1.1
mvn versions:set -f pom.xml -s nexus_settings.xml -
DgenerateBackupPoms=false -DnewVersion=${VERSION}
git add pom.xml src/main/webapp/index.jsp
git commit -m "Increased version to ${VERSION}"
git push private master
```

2. Validate that this push triggered a new build in Jenkins.
3. After a few minutes, when prompted by the pipeline, approve the switch to the new version.
   a. Click **Proceed**.
4. Once the pipeline finishes successfully, verify that you can see the updated application.

---

6. Build Configuration with Pipeline

Instead of defining the pipeline in Jenkins, you can create an OpenShift build configuration with a Pipeline build strategy. This build configuration must be in the same project as the Jenkins pod (unless you configure the `master-config.yaml` to point to another Jenkins instance).

1. To integrate the pipeline with the OpenShift web console, create a build configuration that points to the `Jenkinsfile` file in your `openshift-tasks-private` repository.
2. You need to create a YAML/JSON file containing your build configuration definition and then create the build configuration from that file.
   - Remember that your `openshift-tasks-private` repository is private, which means OpenShift needs credentials to access the `Jenkinsfile` file.
3. In the OpenShift web console, switch to your Jenkins project and navigate to **Builds** → **Pipelines**.
4. Click **Start Pipeline** to trigger a new pipeline.
5. Click **View Log** to view the pipeline progression and follow along in Jenkins.

---

7. Clean Up Environment

1. To clean up the environment, delete all of your projects:

```
oc delete project xyz-tasks-dev
oc delete project xyz-tasks-prod

oc delete project xyz-gogs
oc delete project xyz-nexus
oc delete project xyz-sonarqube
oc delete project xyz-jenkins
```

Build Version: 0370f14b71b9454d6e9f9f526dc59f5c8afe123c : Last updated 2018-03-13 11:12:45 EDT