Table of Contents

Application Development Lab

In this lab, you examine more complex build configurations. You build a project from a protected repository that requires setting up access to the repository. You also examine binary builds and chained builds, keeping the resulting container image small.

Goals

- Set up build configurations
- Implement binary builds
- Implement chained builds

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 Build Configurations 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 Build Configurations

2.1. Create Build Secrets

1. Create a new project named `xyz-builds` with the display name `XYZ Builds`, replacing `xyz` with your initials:

   ```
   oc new-project xyz-builds
   ```

2. Log in to Gogs and switch to the organization named `CICDLabs`. If you do not have this organization, create it.
3. Under the `CICDLabs` organization, create a new repository named `openshift-tasks-private` and make sure it is set to `Private`.
   a. Make a note of the Gogs repository path.
      - Expect it to resemble `http://<gogs-route>/CICDLabs/openshift-tasks-private.git`.

   b. Make a copy of the `openshift-tasks` repository and push it into Gogs:

   ```
   cd $HOME
   git clone https://github.com/wkulhanek/openshift-tasks.git
   cd $HOME/openshift-tasks
   git remote add private <repository path>
   git push private master
   ```

   Expect to be prompted for your Gogs user ID and password.

4. Create a new application pointing to this repository:

   ```
   oc new-app --template=eap70-basic-s2i --param
   APPLICATION_NAME=tasks --param
   SOURCE_REPOSITORY_URL=http://gogs.xyz-
   gogs.svc.cluster.local:3000/CICDLabs/openshift-tasks-private.git
   --param SOURCE_REPOSITORY_REF=master --param CONTEXT_DIR=/ --
   param MAVEN_MIRROR_URL=http://nexus3-xyz-
   nexus.apps.na37.openshift.opentlc.com/repository/maven-all-public
   ```

Replace `xyz-gogs` with your Gogs project name. Also make sure your `MAVEN_MIRROR_URL` points to your specific Nexus instance. If you do not have a Nexus instance yet, you can omit the last parameter.

5. Review the logs and note that a build error is displayed:

```
$ oc logs -f tasks-1-build
Error from server (BadRequest): container "sti-build" in pod
"tasks-1-build" is waiting to start: PodInitializing
```

6. Investigate further to determine why the build is failing:

```
oc describe pod tasks-1-build
```

Sample Output

```
[...]
    State:      Terminated
      Reason:   Error
      Message:  Cloning "http://gogs.xyz-
gogs.svc.cluster.local:3000/CICDLabs/openshift-tasks-private.git"
...
error: failed to fetch requested repository "http://gogs.xyz-
gogs.svc.cluster.local:3000/CICDLabs/openshift-tasks-private.git"
with provided credentials

      Exit Code:        1
[...]
```

7. Create an appropriate secret to hold the credentials used to access the Gogs (GitHub) repository.

```
oc secrets new-basicauth gogs-secret --username=<user_name> --
password=<password>
```

8. Associate the build secret with the build configuration.

```
oc set build-secret --source bc/tasks gogs-secret
```

9. Start a new build to pick up these credentials:

```
oc start-build tasks
```

- Expect this build to finish successfully.

2.2. Cache Artifacts

The JBoss EAP S2I builder image supports saving build artifacts between builds, which dramatically reduces build times. The build configuration needs to reflect that.

1. Change your build configuration to build incrementally. You may also want to change **`forcePull`** to **`false`**. This prevents OpenShift from pulling the builder image every single time it builds the application, which takes a lot of time.

   a. Use **`oc patch`** to edit the build configuration and add the **`incremental`** flag under **`sourceStrategy`** as well as change **`forcePull`** to **`false`**:

   ```
   oc patch bc/tasks --patch='{"spec": {"strategy":
   {"sourceStrategy": {"incremental": true}}}}'
   oc patch bc/tasks --patch='{"spec": {"strategy":
   {"sourceStrategy": {"forcePull": false}}}}'
   ```

2. Start another build and examine the build logs to make sure the change was applied:

   ```
   oc start-build tasks
   oc logs -f tasks-3-build
   ```

3. Implement Binary Builds

In this section, you use the OpenJDK S2I image to demonstrate binary builds with an existing Spring Boot application.

3.1. Create Spring Boot Application

1. Create a new Java Spring Boot application from https://github.com/wkulhanek/ola.git (https://github.com/wkulhanek/ola.git).
   - Use the **`redhat-openjsk18-openshift`** image stream with the **`1.2`** tag to build the application.
   - Make sure to create a route for the application after it has been created.

```
oc new-app --image-stream=redhat-openjdk18-openshift:1.2
https://github.com/wkulhanek/ola.git
oc expose svc ola
```

2. Make sure your application is running:

```
curl http://$(oc get route ola --template='{{ .spec.host
}}')/api/ola
```

3.2. Deploy Using Binary Build

In this section, you build the same application using the binary build strategy, which means you first build the application locally (in your client VM), then create a binary build configuration, and finally start a binary build using the locally built JAR file as the input to the binary build.

1. Build the application locally:

```
cd $HOME
git clone https://github.com/wkulhanek/ola.git
cd ola
mvn clean package
```

   - This creates the `$HOME/ola/target/ola.jar` file.

2. Start the application:

```
java -jar $HOME/ola/target/ola.jar
```

3. Once the application starts, test the application from another terminal window:

```
curl http://127.0.0.1:8080/api/ola
```

4. Create a binary build called `ola-binary`.

```
oc new-build --binary=true --name=ola-binary --image-
stream=redhat-openjdk18-openshift:1.2
```

   - This build now expects the binary deployment artifact from the local file system.

5. Start a new build and stream the compiled file into the build. Make sure you follow the build as it executes.

```
oc start-build ola-binary --from-file=$HOME/ola/target/ola.jar --
follow
```

Expect to see the build finish very quickly when you execute the `oc start-build` command. Binary build copies a pre-built artifact and moves the copy into the correct directory. In this case, it copies the `ola.jar` file into the S2I image and then moves it into `/deployments`.

6. Once the build finishes, deploy the application from the newly created image.
   - Remember to expose the application as the `ola-binary` route.

     ```
     oc new-app ola-binary
     oc expose svc/ola-binary --port=8080
     ```

7. When the application is running (check the logs), make sure to test it:

   ```
   curl http://$(oc get route ola-binary --template='{{ .spec.host
   }}')/api/ola
   ```

4. Implement Chained Builds

Chained builds can be used to keep the runtime image small and free of any compile tools or artifacts. First you build the application using a builder image. Then you take the built artifact and deploy it into a runtime image. Obviously, this only makes sense for compiled languages like Java or Go.

In this section, you create a chained build using a Go builder image and a second image for the actual runtime. Then you deploy an application based on the second image.

1. Import the `jorgemoralespou/s2i-go` image from DockerHub for use as your S2I Go image.

   ```
   oc import-image jorgemoralespou/s2i-go --confirm
   ```

   Sample Output

```
The import completed successfully.

Name:                   s2i-go
Namespace:              xyz-builds
Created:                Less than a second ago
Labels:                 <none>
Annotations:
openshift.io/image.dockerRepositoryCheck=2018-02-09T21:38:34Z
Docker Pull Spec:       docker-
registry.default.svc:5000/chained/s2i-go
Image Lookup:           local=false
Unique Images:          1
Tags:                   1

latest
  tagged from jorgemoralespou/s2i-go
[...]
```

2. Create a new build to compile the sample Go application.
   - Name the build `builder`.
   - Use the `s2i-go` builder image.
   - The application source code is at https://github.com/tonykay/ose-chained-builds (https://github.com/tonykay/ose-chained-builds).
   - The context directory to use is `/go-scratch/hello_world`.

   ```
   oc new-build s2i-go~https://github.com/tonykay/ose-chained-
   builds \
      --context-dir=/go-scratch/hello_world --name=builder
   ```

3. Follow the build logs and wait for the Go application to compile and the image to finish building before moving on to the next step.
4. Examine the resulting `builder imagestream`:

```
oc describe is builder
```

Sample Output

```
Name:                   builder
Namespace:              xyz-builds
Created:                7 minutes ago
Labels:                 build=builder
Annotations:            openshift.io/generated-
by=OpenShiftNewBuild
Docker Pull Spec:       docker-registry.default.svc:5000/xyz-
builds/builder
Image Lookup:           local=false
Unique Images:          1
Tags:                   1

latest
  no spec tag

  * docker-registry.default.svc:5000/xyz-
builds/builder@sha256:3e641f5a5517621d7ddb5a40bf8fffc52b95c74c58d
e84674199902e9df3f91d
      6 minutes ago
```

5. Create the second (chained) build that takes the built artifact and deploys it into a small runtime image.
   - Name the build **`runtime`**.
   - Use the **`scratch`** Docker image as the base image.
   - Use your **`builder`** image as a source image.
   - The generated artifact in the builder image is located at **`/opt/app-root/src/go/src/main/main`**.
   - Use the Docker build strategy with this inline Dockerfile: **`FROM scratch\nCOPY main /main\nEXPOSE 8080\nENTRYPOINT ["/main"]`**.

   ```
   oc new-build --name=runtime \
      --docker-image=scratch \
      --source-image=builder \
      --source-image-path=/opt/app-root/src/go/src/main/main:. \
      --dockerfile=$'FROM scratch\nCOPY main /main\nEXPOSE
   8080\nENTRYPOINT ["/main"]'
   ```

6. Follow the build process:

```
oc logs -f bc/runtime
```

7. Deploy and expose the application once it is built:

```
oc new-app runtime --name=my-application
oc expose svc/my-application
```

8. Examine the exposed application in your browser or with `curl`:

```
curl $(oc get route my-application --template='{{ .spec.host }}')
```

5. Clean Up Environment

1. Delete your project:

```
oc delete project xyz-builds
```