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

## 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

1. Log in to Gogs and switch to the organization named **CICDLabs**. If you do not have this organization, create it.
2. Under the **CICDLabs** organization, create a new repository named **openshift-tasks-private** and make sure it is set to **Private**.
   1. Make a note of the Gogs repository path.
      * Expect it to resemble **http://<gogs-route>/CICDLabs/openshift-tasks-private.git**.
   2. Make a copy of the **openshift-tasks** repository and push it into Gogs:
   3. cd $HOME
   4. git clone https://github.com/wkulhanek/openshift-tasks.git
   5. cd $HOME/openshift-tasks
   6. git remote add private <repository path>

git push private master

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

1. 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.svc.cluster.local:8081/repository/maven-all-public

|  |  |
| --- | --- |
|  | Replace **xyz-gogs** with your Gogs project name and **xyz-nexus** with your Nexus project name. If you do not have a Nexus instance yet, you can omit the last parameter. |

1. Review the logs and note that a build error is displayed:
2. $ oc logs -f tasks-1-build

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

1. 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

[...]

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

oc create secret generic gogs-secret --from-literal=username=<user\_name> --from-literal=password=<password>

1. Associate the build secret with the build configuration.

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

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

oc start-build tasks

* 1. 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.
   1. Use **oc patch** to edit the build configuration and add the **incremental** flag under **sourceStrategy** as well as change **forcePull** to **false**:
   2. oc patch bc/tasks --patch='{"spec": {"strategy": {"sourceStrategy": {"incremental": true}}}}'

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

1. Start another build and examine the build logs to make sure the change was applied:
2. 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>.
   * Use the **redhat-openjdk18-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

1. 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:
2. cd $HOME
3. git clone https://github.com/wkulhanek/ola.git
4. cd ola

mvn clean package

|  |  |
| --- | --- |
|  | This build does not use a Maven Mirror. If you still have a **nexus\_settings.xml** file from a previous build you may use it by adding `-s <path\_to>/nexus\_settings.xml to your command line. |

* + This creates the **$HOME/ola/target/ola.jar** file.

1. Start the application:

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

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

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

1. 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.

1. 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**. |

1. 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

1. 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

[...]

1. 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>.
   * 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

1. 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.
2. 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@sha256:3e641f5a5517621d7ddb5a40bf8fffc52b95c74c58de84674199902e9df3f91d

6 minutes ago

1. 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 \
   * --source-image=builder \
   * --source-image-path=/opt/app-root/src/go/src/main/main:. \

--dockerfile=$'FROM scratch\nCOPY main /main\nEXPOSE 8080\nENTRYPOINT ["/main"]'

1. Follow the build process:

oc logs -f bc/runtime

1. Deploy and expose the application once it is built:
2. oc new-app runtime --name=my-application

oc expose svc/my-application

1. 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

应用开发实验室

在本实验中，您将研究更复杂的构建配置。您从受保护的存储库构建项目，该存储库需要设置对存储库的访问权限。您还可以检查二进制构建和链接构建，从而使生成的容器图像保持较小。

# 目标

* **设置构建配置**
* **实现二进制构建**
* **实现链式构建**