



VMware Advanced Customer Engagements (ACE) Team

Omnibus GitLab Integration with VMware TKGI and Harbor

How-to guide for Integration and Configuration of CI/CD with
examples

MAY 2020

Table of Contents

1. Introduction.....	2
2. Pre-requisites	2
3. Add Existing TKGI K8s Cluster to GitLab project.....	3
4. Install and configure GitLab Runner using Helm chart, associate it with project	7
5. Configure and Execute CI/CD Pipeline for Building, Tagging and Publishing Application Image into Registry	13
6. Configure and Execute CI/CD Pipeline for Deployment of Application from Registry to K8s Cluster	20

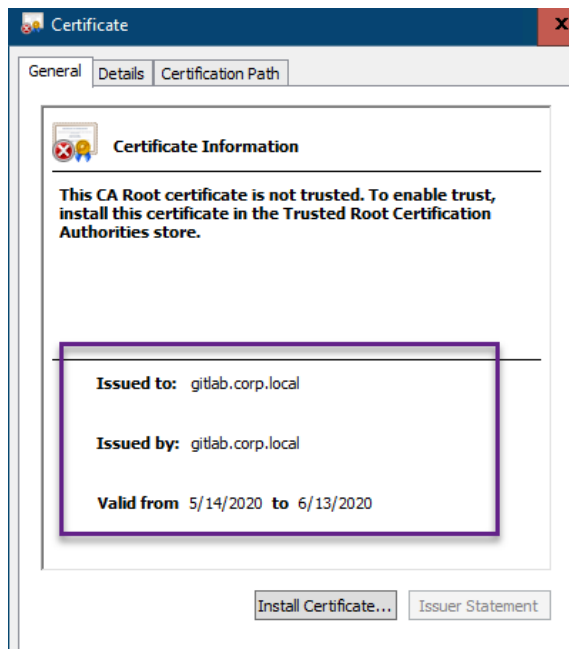
1. Introduction

In this document, we provide an overview of integration of OmniBus GitLab (Enterprise Edition) software change management (SCM) platform with VMware Tanzu Kubernetes Grid Integrated (TKGI, formerly known as VMware Enterprise PKS) Kubernetes clusters and Harbor container image registry platform for automated software build tasks. GitLab is a popular DevOps platform since it is compatible with Git file versioning, project directory structure and client software.

We highlight configuration steps to enable integration between GitLab EE and Kubernetes clusters provisioned with TKGI platform and Harbor container image registry to enable CI/CD process automation using GitLab tools.

2. Pre-requisites

- The following software should be installed and accessible from
 - VMware TKGI (formerly 'Enterprise PKS', v 1.6.1 or 1.7)
 - Kubernetes cluster provisioned via TKGI environment that is accessible via kubectl CLI
 - OmniBus GitLab Enterprise Edition (v 11.2 or later, v 12.10.5-ee used in a Lab for this paper) installed and configured, accessible via URL like <https://gitlab.corp.local> via administrator level account
 - A GitLab project that contains software artifacts that can be built into container images. We are using the following example from GitHub:
<https://github.com/riazvm/dockersample> cloned into a local GitLab project
 - "CLI VM" – typically a Linux VM that is used for Command Line access to Kubernetes clusters and runs other tools (Docker, Helm etc.) for configuration of integrations and intermediate validation of build process stages.
- There should be no networking issues (firewalls, blocked ports, DNS resolution etc.) between GitLab VM, TKGI K8s clusters and Harbor VM
- Main GitLab URL (such as **gitlab.corp.local**) should have a valid CA certificate (typically generated by GitLab installer script such as shown below:



3. Add Existing TKGI K8s Cluster to GitLab project

We need to add K8s cluster to our GitLab project as a target for CI/CD deployments of containerized applications and for deploying Runner components that execute pipeline tasks/scripts. See GitLab [documentation](#) for more information on Runners

- Start with “Add Existing Cluster” Tab in the “Operations – Kubernetes” menu for a project:

Specify name and FDQN (create DNS record if doesn't exist yet) based URL of API Server/Master node(s)

Did you know?

Every new Google Cloud Platform (GCP) account receives \$300 in credit upon [sign up](#). In partnership with Google, GitLab is able to offer an additional \$200 for both new and existing GCP accounts to get started with GitLab's Google Kubernetes Engine Integration.

[Apply for credit](#)

Add a Kubernetes cluster integration

With a Kubernetes cluster associated to this project, you can use review apps, deploy your applications, run your pipelines, and much more in an easy way.

[Learn more about Kubernetes](#)

Create new cluster on GKE

Add existing cluster

Enter the details for your Kubernetes cluster

Please enter access information for your Kubernetes cluster. If you need help, you can read our [documentation](#) on Kubernetes

Kubernetes cluster name

daniel-lab2-small1-sharedt1.corp.local

API URL

https://daniel-lab2-small1-sharedt1.corp.local:8443

- Follow “More Information” links for each field to be filled in, as specified in documentation https://gitlab.acelab.local/help/user/project/clusters/add_remove_clusters.md#add-existing-cluster with the following fields:

Obtain CA certificate from the K8s cluster using command like

```
kubectl get secret <secret name> -o jsonpath =
"{'data':{'ca\.crt'}}" | base64 -decode
```

E.G. `kubectl get secret <default-token-p6br2> -o jsonpath =`
`"{'data':{'ca\.crt'}}" | base64 -d`

Note: If the command returns the entire certificate chain, copy the *root ca* certificate value at the bottom of the chain:

-----BEGIN CERTIFICATE-----

```
MIIC+zCCAeOgAwIBAgIUBEYdVQpHO7z4r608A+8wNRLmhbkwDQYJKoZIhvcNAQEL
BQAwDTELMAkGA1UEAxMCY2EwHhcNMjAwNDIyMDQ1NDM1WhcNMjAwNDIyMDQ1NDM1
WjANMQswCQYDVQQDEwJjYTCASlwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEB
```

.....


```
C8h+Hip2IxIN/Kubq7Hv3yNFD9MbtpCRmP9nFCo/UFapjljvtd6O0F1qPOQzU8=
```

-----END CERTIFICATE-----

Copy the above certificate string value for use in the following steps

- Obtain Authentication Token for GitLab authentication against K8s

GitLab authenticates against K8s using service tokens, which are scoped to a namespace. The token used should belong to a service account with ‘cluster-admin’ privileges.

Follow GitLab [documentation](#) to create a Service Account and Cluster Role Binding with “cluster-admin” privileges using sample *gitlab-admin-service-account.yaml* K8s deployment descriptor provided as an example:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: gitlab-admin
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: gitlab-admin
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: gitlab-admin
  namespace: kube-system
```

Create Service account and Cluster Role Binding in the target cluster:

```
kubectl apply -f gitlab-admin-service-account.yaml
serviceaccount/gitlab-admin created
clusterrolebinding.rbac.authorization.k8s.io/gitlab-admin created
```

- **Retrieve token for the *gitlab-admin* Service Account:**

```
kubectl -n kube-system describe secret $(kubectl -n kube-
system get secret | grep gitlab-admin | awk '{print $1}')
```

Name: gitlab-admin-token-rf6fr
Namespace: kube-system
Labels: <none>
Annotations: kubernetes.io/service-account.name: gitlab-admin
kubernetes.io/service-account.uid: dd65123d-1a2c-47e7-8a5e-97adf871c27d

Type: kubernetes.io/service-account-token

Data

====

ca.crt: 1094 bytes

namespace: 11 bytes

token: < authentication-token >

Paste value of **authentication-token** into the 'Service-Token' field in the "Add Existing Cluster" GitLab screen:

API URL

The URL used to access the Kubernetes API. [More information](#)

CA Certificate

```
hjUiGoOV9/lxsQyxWV/8UkFlzF/msgP6DqbLDtW2AjeZIGwWw9zfpiQvQeAtE9
VtQQheGUodxBd+o9TCBdGOKa3CsZJ8amziaEo6UdkO1Tyuir6YebTucWi7HYbGT
xednvnWmPQSlpVnjpYntkkCAwEAAANTMFEwHQYDVR0OBBYEFBWHvBU3vWY8IM
Rs5FZaSJVTHwMB8GA1UdIwQYMBaAFBWHvBU3vWY8IMRs5FZaSJVTHwMA8GA1Ud
EwEB/wQFMAMBAf8wDQYJKoZIhvcNAQELBQADggEBAISm0S29TPfUEBXnYdHYPa8
jTl3hLajfjAK2PWdEwaCPTggRV867TMX5N7Te7b7S5YWvycQerHcSzROxIsUpI
UVEtZCu/RxEV4JbDjh6o2cKF1L/r67MzK8JRam5GxrfP+e2dPmThHBldrJ8RkI
bqmk0z8b5I3OTJvZwC+Cq66BARS1/XweyEP4XyIBPFQE292WwA6Ktw60j3sCkp/O
Rj01FgiS2alhQVtgpI3VaCj6Y5431NH23tJ0yS8KQLzQNqjIVggk+X1AvzvRNIMh
C8h+Hip2lXN/Kubq7Hv3yNFD9MbtncRmP9nFCo/UFapjIjvtd6O0F1qPOQzU8=
-----END CERTIFICATE-----
```

The Kubernetes certificate used to authenticate to the cluster. [More information](#)

Service Token

A service token scoped to `kube-system` with `cluster-admin` privileges. [More information](#)

☐ RBAC-enabled cluster

Enable this setting if using role-based access control (RBAC). This option will allow you to install applications on RBAC clusters. [More information](#)

☐ GitLab-managed cluster

Allow GitLab to manage namespace and service accounts for this cluster. [More information](#)

Project namespace prefix (optional, unique)

- Click “Add Cluster” button – should get a confirmation of successful result on GitLab UI:

i Kubernetes cluster was successfully updated.
 ×

Project cluster **daniel-lab2-small1-sharedt1.corp.local**

Details
 Health
 Applications
 Advanced Settings

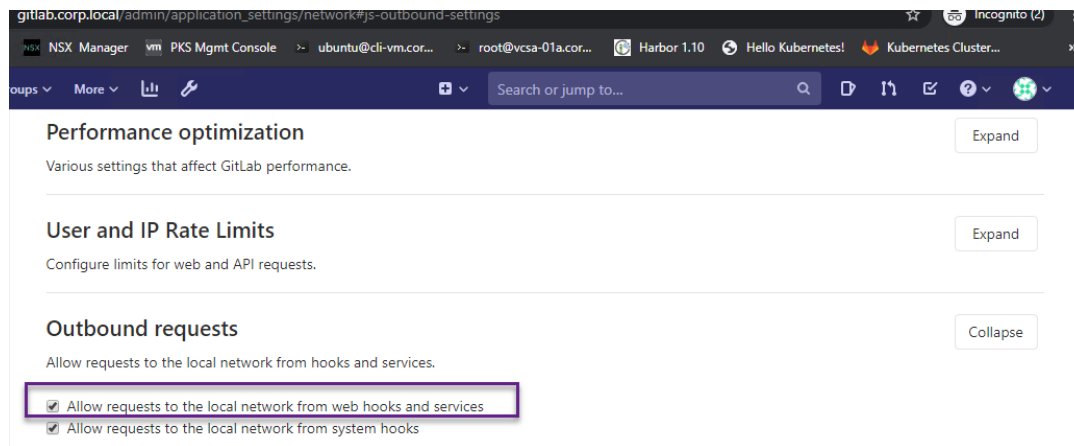
GitLab Integration ✓

Environment scope

*

* is the default environment scope for this cluster. This means that all jobs, regardless of their environment, will use this cluster. [More information](#)

NOTE: in case if a warning about blocked requests to local networks is displayed (when GitLab VM and K8s cluster API are on the same network), we may need to explicitly allow requests to local networks from Web Hooks and Services, following KB Article: <https://gitlab.com/gitlab-org/gitlab-foss/-/issues/57948>



4. Install and configure GitLab Runner using Helm chart, associate it with project

In GitLab CI/CD, Runners run the code defined in the `.gitlab-ci.yml` pipeline definition file. They can be dedicated virtual machines or dedicated Kubernetes Pods that pick up build jobs through the coordinator API of GitLab CI/CD. A Runner can be specific to a certain project or serve any project in GitLab CI/CD, the latter is called a Shared Runner.

Below are the steps to prepare for installation of GitLab Runners as in-cluster K8s resource via Helm chart, performed on CLI VM.(generally, follows the documentation <https://docs.gitlab.com/runner/install/kubernetes.html>)

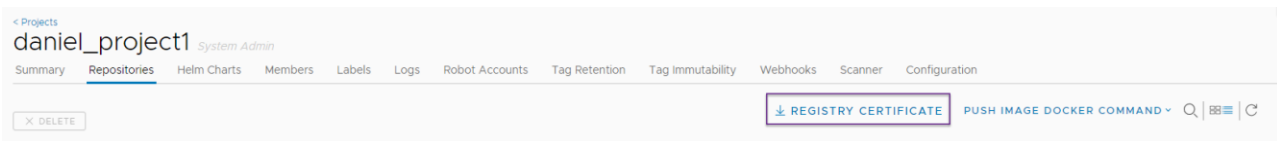
- Install Helm client/server following [documentation](#) , validate that it can reach general repositories containing **gitlab-runner** charts

E.G. `helm search hub gitlab-runner`

URL	CHART VERSION	APP VERSION	DESCRIPTION
https://hub.helm.sh/charts/choerodon/gitlab-runner	0.2.4	0.2.4	gitlab-runner for Choerodon
https://hub.helm.sh/charts/pnnl-miscscripts/gitlab-runner	0.1.3	0.1.2-1	A Helm chart for Kubernetes
https://hub.helm.sh/charts/camptocamp/gitlab-runner	0.12.6	12.6.0	GitLab Runner
https://hub.helm.sh/charts/gitlab/gitlab-runner	0.16.0	12.10.1	GitLab Runner

- Download Harbor Registry certificate from its UI

Login to Harbor UI, navigate to the Project where plan to host built container images click on “Registry certificate” to download the certificate file:



- Create namespace in the K8s cluster where GitLab Runner Pod will be running

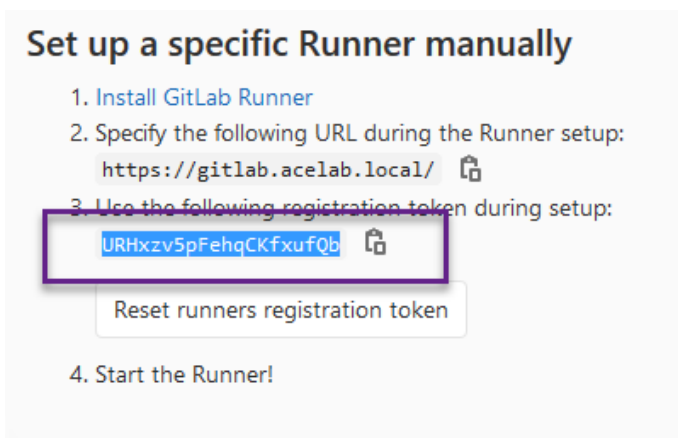
```
kubectl create ns gitlabrunner
```

NOTE: Here the Runner namespace is called **gitlabrunner** but it can be other valid namespace name

Set that namespace as current context:

```
kubectl config set-context --current --namespace=gitlabrunner
```

- Navigate in GitLab UI to “Settings → CI/CD → Runners”:



and copy values for GitLab URL and Runner Registration token from the screen above.

- Use GitLab URL and registration token values obtained in the previous step in the **values.yaml** Helm chart configuration file (full example available for Runner Helm chart installation in the GitHub repository: <https://gitlab.com/gitlab-org/charts/gitlab-runner/-/blob/master/values.yaml>)

```
## GitLab Runner Image
##
## By default it's using gitlab/gitlab-runner:alpine-v{VERSION}
## where {VERSION} is taken from Chart.yaml from appVersion field
##
## ref: https://hub.docker.com/r/gitlab/gitlab-runner/tags/
```

```
##
##image: gitlab/gitlab-runner:alpine-v11.6.0
gitlabUrl: https://gitlab.acelab.local
runnerRegistrationToken: 'URHxzv5pFehqCKfxufQb'
```

NOTE: please see GitLab documentation
<https://docs.gitlab.com/runner/install/kubernetes.html> for recommended values of additional fields in values.yaml file for GitLab Runner Helm chart.

- Another important field is RBAC support for a Runner service account. To have the chart create new Service account during installation, set **rbac.create** to **true**

For RBAC support:

rbac:

create: true

Define specific rbac permissions.

resources: ["pods", "pods/exec", "secrets"]

verbs: ["get", "list", "watch", "create", "patch", "delete"]

..

(Otherwise, set to **rbac create** to **false** and specify existing Service Account)

- An important setting is Max number of concurrent jobs to run which is controlled by **concurrent** filed value. Set it based on projected size of build jobs and related resource utilization:

..

Configure the maximum number of concurrent jobs

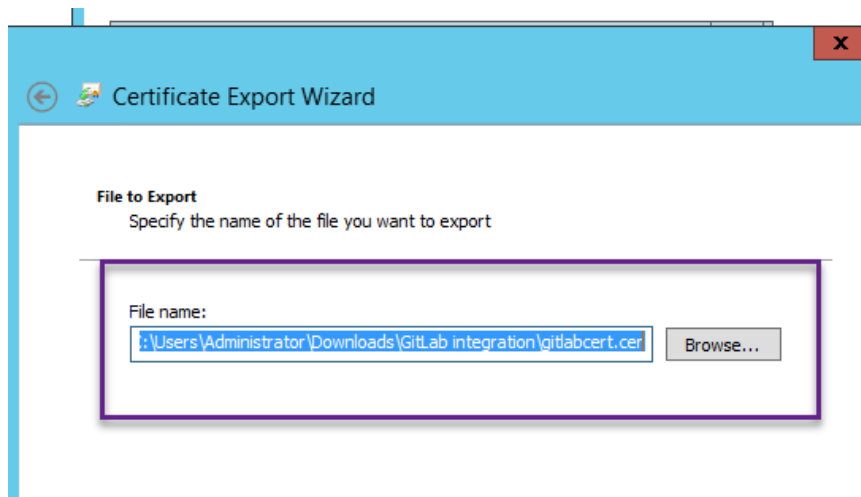
ref: <https://docs.gitlab.com/runner/configuration/advanced-configuration.html#the-global-section>

##

concurrent: 10

..

- To allow containers activated on Runner to make API calls against GitLab Secure API, we need to export SSL certificate from GitLab server in a CER (BASE 64) format into a file:



- ✓ The certificate file name used should be in the format **<gitlab.hostname.domain.crt>**, for example **gitlab.corp.local.crt**.
 - ✓ Any intermediate certificates need to be concatenated to your server certificate in the same file.
 - ✓ The hostname used should be the one the certificate is registered for.
- Generate K8s secret from GitLab CA certificate file (saved as **gitlab.corp.local.crt** in previous step) that complies with above conditions in the K8s namespace where Runner will be deployed using command like:

```
$ kubectl create secret generic gitlabca --from-
file=gitlab.corp.local.crt -n gitlabrunner
secret/gitlabca created
```

Validate the secret got created

```
$ kubectl get secrets
```

NAME	TYPE	DATA	AGE
default-token-ckt9m	k10ubernetes.io/service-account-token	3	30m
gitlabca	Opaque	1	9s

- Use that K8 Secret name in the **certsSecretName** section of the **values.yaml** file as shown below:

```
..
## Set the certsSecretName in order to pass custom certificates for GitLab Runner to use
## Provide resource name for a Kubernetes Secret Object in the same namespace,
## this is used to populate the /home/gitlab-runner/.gitlab-runner/certs/ directory
```

```
## ref: https://docs.gitlab.com/runner/configuration/tls-self-signed.html#supported-
options-for-self-signed-certificates
## secret name
certsSecretName: gitlabca
```

- Configure environment variables that will be present when Runner registration command runs in the following section of values.yaml file:

```
## This provides further control over the registration process and the config.toml file
## ref: `gitlab-runner register --help`
## ref: https://docs.gitlab.com/runner/configuration/advanced-configuration.html
##
envVars:
  - name: RUNNER_ENV
    value: "DOCKER_TLS_CERTDIR="
  - name: CI_SERVER_TLS_CA_FILE
    value: /home/gitlab-runner/.gitlab-runner/certs/gitlab.acelab.local.crt
```

NOTE: certificate file name in the value for **CI_SERVER_TLS_CA_FILE** variable should be same as file name used to generate K8s secret above (**gitlabca**) used in the **certsSecretName** section

- If CI/CD task will require using “executor” images running containers in ‘privileged’ mode (such as when using popular DIND “docker in docker”, per GitLab [documentation](#)), perform the following optional configuration steps:
 - a. Update “privileged” parameter flag in the **values.yaml** file:

```
## Run all containers with the privileged flag enabled
## This will allow the docker:dind image to run if you need to run Docker
## commands. Please read the docuvi s before turning this on:
## ref: https://docs.gitlab.com/runner/executors/kubernetes.html#using-docker-
dind
##
privileged: true
....
```

- b. Copy previously downloaded Harbor certificate file to **/etc/gitlab/trusted-certs** and **/etc/gitlab/ssl** folders in GitLab VM:

```
ls /etc/gitlab/ssl
ca_harbor.crt
gitlab.acelab.local.crt
....
ls /etc/gitlab/trusted-certs
ca_harbor.crt
```

```
gitlab.acelab.local.crt
```

c. Target TKGI K8s cluster should be deployed with Pod Security Policies set to “privileged” mode, per [documentation](#)

NOTE: in our CI/CD example below we will be using an executor container image based on Google Project Kaniko which **does not require running in privileged mode**, please see GitLab [documentation](#) for details. Therefore, the configuration steps in this section are not required for running that example.

- Add a Helm repository containing the chart for Runner deployment:

```
helm repo add stable https://kubernetes-  
charts.storage.googleapis.com
```

Verify that repository has been added and is available:

```
helm repo list  
NAME URL  
gitlab https://charts.gitlab.io  
stable https://kubernetes-charts.storage.googleapis.com
```

- Perform a Runner installation using Helm chart, from the directory where **values.yaml** file is located:

```
helm install gitlab-runner -f ./values.yaml gitlab/gitlab-  
runner -n gitlabrunner
```

NOTES:

- ✓ IMPORTANT: if running Helm command from another directory that doesn't contain **values.yaml** file, provide full path to that file to customize
- ✓ 'gitlab-runner' is the name of Runner chart deployment, chosen arbitrary
- ✓ gitlab/gitlab-runner is the name of Helm chart from the repository

An output of 'helm install' command should look like:

```
NAME: gitlab-runner  
LAST DEPLOYED: Sun May 3 05:46:50 2020  
NAMESPACE: gitlabrunner  
STATUS: deployed  
REVISION: 1  
TEST SUITE: None  
NOTES:  
Your GitLab Runner should now be registered against the GitLab instance  
reachable at: https://gitlab.acelab.local
```

- (Optional) validate that Runner deployments/pods are running in the designated K8s namespace:

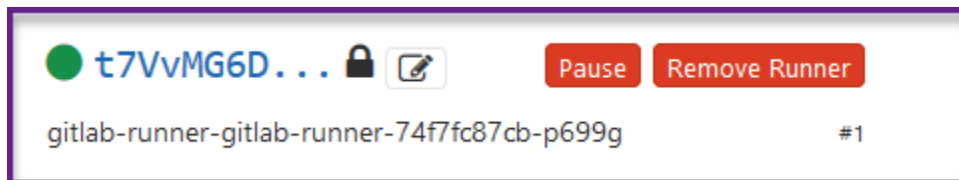
```
kubectl get deploy, po -n gitlabrunner
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/gitlab-runner-gitlab-runner	1/1	1	1	33s

NAME	READY	STATUS	RESTARTS	AGE
pod/gitlab-runner-gitlab-runner-74f7fc87cb-p699g	1/1	Running	0	33s

- Verify that newly installed Runner is configured for GitLab project(s):

Navigate to **Settings** → **CI/CD** for the Project and check whether Runner shows as active:



Notes:

- ✓ Usually, Runner configured on a project level will show up in its Settings → CI/CD automatically, when deployed to integrated cluster. In other cases they may be additional steps needed to make Runner available for a GitLab project, per [documentation](#)
- ✓ Same instance GitLab Runner can be optionally shared among multiple projects, that can be configured in GitLab “CI/CD Settings → Shared Runners”, per [documentation](#)

5. Configure and Execute CI/CD Pipeline for Building, Tagging and Publishing Application Image into Registry

Below is an example of a CI/CD pipeline that builds a simple SpringBoot microservice from its Java source code using Maven, continues to build a container image using Dockerfile residing in a subdirectory and finally pushes built image into designated project in the Harbor container image repository.

- Properties of target Harbor project (top level construct for images hosting) are shown below. It is not ‘Public’ and therefore requires authorized user login with at least “Developer” access level, per Harbor [documentation](#). It has image vulnerability scanning on ‘push’: any time an image is added to a registry via ‘docker push...’ command, it will be scanned for vulnerabilities automatically.

Project registry

☐ Public

Making a project registry public will make all repositories accessible to everyone.

Deployment security

☐ Enable content trust

Allow only verified images to be deployed.

☐ Prevent vulnerable images from running.

Prevent images with vulnerability severity of **Medium** and above from being deployed.

Vulnerability scanning

☒ Automatically scan images on push

Automatically scan images when they are pushed to the project registry.

CVE whitelist

Project whitelist allows vulnerabilities in this list to be ignored in this project when pushing and pulling images.

You can either use the default whitelist configured at the system level or click on 'Project whitelist' to create a new whitelist

Add individual CVE IDs before clicking 'COPY FROM SYSTEM' to add system whitelist as well.

☒ System whitelist ☐ Project whitelist

ADD COPY FROM SYSTEM

None

Expires at

Never expires

☒ Never expires

- There are existing image repositories configured for that project, shown below:

Name	Tags	Pulls
daniel_project1/adbs-sync	2	8
daniel_project1/app-server	2	6
daniel_project1/dockersample	2	3
daniel_project1/frontend	2	6
daniel_project1/mysql	1	1

Our GitLab pipeline will be building and pushing images into 'daniel_project1/dockersample' repository:

Tag	Size	Pull Command	Vulnerabilities	Signed	Author	Creation Time	Docker Version	Labels
latest	80.53MB		11 Total • 11 Fixable			5/19/20, 12:47 PM	1.12.6	

There is an existing version of an image

- Structure of the example project **dockersample** is shown below:

Name	Last commit	Last update
qrcode	Initial seed from cloned GitHub repo to test CI/CD pipeline	1 day ago
.gitlab-ci.yml	Update .gitlab-ci.yml added use of CI_REGISTRY_TAG variable. NB! Use ...	22 hours ago
README.md	Initial seed from cloned GitHub repo to test CI/CD pipeline	1 day ago
gitlab-ci-docker.yaml	Initial seed from cloned GitHub repo to test CI/CD pipeline	1 day ago

dockersample - QRCodeGenerator

- Add default CI/CD pipeline script file (**.gitlab-ci.yml**) at the root level of the Project using “CI/CD Configuration” option

NOTE: **.gitlab-ci.yml** CI/CD script syntax should comply with structure and stages defined in accordance with documentation: <https://docs.gitlab.com/ee/ci/yaml/README.html>

- Edit contents of that script file from GitLab IDE (or outside of it and use Git client to commit changes to project repository):

Commit message: Update .gitlab-ci.yml

Target Branch: master

Commit changes

NOTES:

- ✓ **.gitlab-ci.yml** is a default CI/CD pipeline file name for any project, other pipeline definitions can be invoked from it

- ✓ Environment variables values referenced in pipeline scripts (CI_REGISTRY, CI_REGISTRY_USER etc.) should be set via “Settings → CI/CD → Variables” section of the project:

Type	Key	Value	Protected	Masked	Environments	
Var	CI_REGISTRY	*****	×	×	All (default)	
Var	CI_REGISTRY_CA_CERT	*****	×	×	All (default)	
Var	CI_REGISTRY_IMAGE	*****	×	×	All (default)	
Var	CI_REGISTRY_PASSWORD	*****	×	×	All (default)	
Var	CI_REGISTRY_TAG	*****	×	×	All (default)	
Var	CI_REGISTRY_USER	*****	×	×	All (default)	

Reveal values Add Variable

Scope of those variables should be normally set as ‘Environment scope’, additional options to protect their values, if required, are available via GitLab settings.

- ✓ For variables that contain values of Container Image registry (Harbor) certificate, make sure it keeps its original format, as shown below:

Key

CI_REGISTRY_CA_CERT

Value

```
-----BEGIN CERTIFICATE-----
MIIDUDCCAjigAwIBAgIUba+VUPLn/L7G/b8wcVVCvdZetWUwDQYJKoZIhvcNAQEL
BQAwHzELMAkGA1UEBhMCVVMxEDAOBgNVBAoMB1Bpdm90YWwwHhcNMjAwNDIxMDAz
NTUzWhcNMjQwNDIyMDAzNTUyWjAfMQswCQYDVQQGEwJVUzEQMA4GA1UECgwHUGI2
b3RhbDCCASlwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAKK+FZOMk0fM3lhj
Biy2o/2GZ2g56KmbFMysc3LHCVP7DqelES62okewTBhEcZET6OfwSx5lFiCkTik
EWNPB:PhutheoWQvY62a+17a/dfo8F1lq/C4HmgnCYuM7TsluTayRmb0F18
-----
```

Type

Var

Environment scope

All (default)

(reason being that we will basically automate commands like: “**docker login \${CI_REGISTRY} -u \${CI_REGISTRY_USER} -p \${CI_REGISTRY_PASSWORD}**” to run as API call from CI/CD script)

- To bypass a need for executor containers to have privileged access to Docker daemon to run “docker build” commands, we can use unprivileged access via [Google Kaniko](https://docs.gitlab.com/ee/ci/) execution environment: <https://docs.gitlab.com/ee/ci/> (OK for running builds, not for running container images) See details in: https://docs.gitlab.com/ee/ci/docker/using_kaniko.html

- Edit default CI/CD pipeline script (**.gitlab-ci.yml**) at the root of GitLab project directory:

master dockersample / +

History Find file Web IDE Clone

Update .gitlab-ci.yml added use of CI_REGISTRY_TAG variable. NB! Use ...
Daniel Zilberman authored 22 hours ago

README CI/CD configuration Add LICENSE Add CHANGELOG Add CONTRIBUTING Add Kubernetes cluster

Name	Last commit	Last update
qrcode	Initial seed from cloned GitHub repo to test CI/CD pipeline	1 day ago
.gitlab-ci.yml	Update .gitlab-ci.yml added use of CI_REGISTRY_TAG variable. NB! Use ...	22 hours ago
README.md	Initial seed from cloned GitHub repo to test CI/CD pipeline	1 day ago
gitlab-ci-docker.yml	Initial seed from cloned GitHub repo to test CI/CD pipeline	1 day ago

README.md

dockersample - QRCodeGenerator

- Example of a working version of above script that builds and pushes Docker container image from GitLab project sub-folders is shown below:

```

1 #added before-script to change directory
2 before_script:
3   - cd $CI_PROJECT_DIR/qrcode
4
5 stages:
6   - build
7   - docker-build
8
9 maven-build:
10  stage: build
11  image: maven:latest
12  script:
13    - mvn package
14
15 docker-package:
16  stage: docker-build
17  image:
18    name: gcr.io/kaniko-project/executor:debug
19    entrypoint: [""]
20  script:
21    - echo "{\"auths\":{\"$CI_REGISTRY\":{\"username\":\"$CI_REGISTRY_USER\",\"password\":\"$CI_REGISTRY_PASSWORD\"}}}" > /kaniko/.docker/config.json
22    - echo $CI_REGISTRY_CA_CERT
23    - echo "-----"
24    - echo "${CI_REGISTRY_CA_CERT}" > /kaniko/ssl/certs/jks-certificates.crt
25    - echo "Copied Harbor Registry CERT into /kaniko/ssl/certs/jks-certificates.crt:"
26    - echo "-----"
27    - cat /kaniko/ssl/certs/jks-certificates.crt
28    - echo "-----"
29    - echo "Project Build Dir:"
30    - echo $CI_PROJECT_DIR
31    - echo "Target Harbor repository image:"
32    - echo $CI_REGISTRY_IMAGE
33    - /kaniko/executor --context $CI_PROJECT_DIR/qrcode --dockerfile $CI_PROJECT_DIR/qrcode/Dockerfile --destination ${CI_REGISTRY_IMAGE}:${CI_REGISTRY_TAG}
34    - echo "CI/CD job completed!"

```

NOTES:

- ✓ Correctly formatted (no TAB characters) section of above **.gitlab-ci.yml** file can be found below:

```

before_script:
  - cd $CI_PROJECT_DIR/qrcode

```

```

stages:
  - test
  - build
  - docker-build
  - deploy

```

```

#Test stage
unit test:

```

```

stage: test
script:
  - echo "Unite Test scripts will go here.."

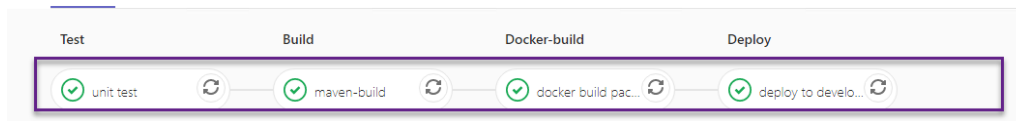
#build stage
maven-build:
  stage: build
  image: maven:latest
  script:
    - mvn package
#docker-build stage
docker build package:
  stage: docker-build
  image:
    name: gcr.io/kaniko-project/executor:debug
    entrypoint: [""]
  script:
    - echo
    "${CI_REGISTRY}":{"username":"${CI_REGISTRY_USER}","password":"${CI_REGISTRY_PASSWORD}"}} > /kaniko/.docker/config.json
    #- echo ${CI_REGISTRY_CA_CERT}
    - echo "-----"
    - echo "${CI_REGISTRY_CA_CERT}" > /kaniko/ssl/certs/jis-certificates.crt
    - echo "Copied Harbor Registry CERT into /kaniko/ssl/certs/jis-certificates.crt:"
    - echo "-----"
    - cat /kaniko/ssl/certs/jis-certificates.crt
    - echo "-----"
    - echo "Project Build Dir:"
    - echo ${CI_PROJECT_DIR}
    - echo "Target Harbor repository image:"
    - echo ${CI_REGISTRY_IMAGE}
    - /kaniko/executor --context ${CI_PROJECT_DIR}/qrqcode --dockerfile ${CI_PROJECT_DIR}/qrqcode/Dockerfile --destination ${CI_REGISTRY_IMAGE}:${CI_REGISTRY_TAG_DEV}
    - echo "CI/CD job completed, APP image should be available at:"
    - echo ${CI_REGISTRY_IMAGE}:${CI_REGISTRY_TAG_DEV}
  environment:
    name: development
#build docker app only when a pipeline on master branch is run
only:
  - master

```

NOTES:

- ✓ This CI/CD script is using **maven:latest** container image for **build** stage jobs execution . It compiles source code using Maven.
- ✓ Then **gcr.io/kaniko-project/executor:debug** container image used for **docker-build** stage jobs execution builds image defined in Dockerfile and pushes it into registry defined by **CI_REGISTRY_IMAGE** environment variable with a tag defined in the **CI_REGISTRY_TAG_DEV**

- ✓ For now, we only want to deploy the application only when pipeline on **master** branch is run hence the 'only ...' condition
- Pipeline execution progress for each stage can be monitored in real time:



and also reviewed after its completion via GitLab CI/CD UI, as shown below:

```

67 $ /kaniko/executor --context ${CI_PROJECT_DIR}/qrcode --dockerfile ${CI_PROJECT_DIR}/Dockerfile --destination ${CI_REGISTRY_IMAGE}:${CI_REGISTRY_TAG_DEV}
68 INFO[0041] Retrieving image manifest java:8-jdk-alpine
69 For verbose messaging see aws.Config.CredentialsChainVerboseErrors
70 INFO[0041] Retrieving image manifest java:8-jdk-alpine
71 INFO[0044] Retrieving image manifest java:8-jdk-alpine
72 INFO[0047] Built cross stage deps: map[]
73 INFO[0047] Retrieving image manifest java:8-jdk-alpine
74 INFO[0047] Retrieving image manifest java:8-jdk-alpine
75 INFO[0050] Executing 0 build triggers
76 INFO[0050] Unpacking rootfs as cmd COPY ./target/qrcode-0.0.1-SNAPSHOT.jar /usr/app/ requires it.
77 INFO[0077] COPY ./target/qrcode-0.0.1-SNAPSHOT.jar /usr/app/
78 INFO[0079] Resolving 1 paths
79 INFO[0079] Taking snapshot of files...
80 INFO[0084] WORKDIR /usr/app
81 INFO[0084] cmd: workdir
82 INFO[0084] Changed working directory to /usr/app
83 INFO[0084] RUN sh -c 'touch qrcode-0.0.1-SNAPSHOT.jar'
84 INFO[0084] Taking snapshot of full filesystem...
85 INFO[0084] Resolving 1672 paths
86 INFO[0088] cmd: /bin/sh
87 INFO[0088] args: [-c sh -c 'touch qrcode-0.0.1-SNAPSHOT.jar']
88 INFO[0088] Running: [/bin/sh -c sh -c 'touch qrcode-0.0.1-SNAPSHOT.jar']
89 INFO[0088] Taking snapshot of full filesystem...
90 INFO[0088] Resolving 1672 paths
91 INFO[0090] ENTRYPOINT ["java", "-jar", "qrcode-0.0.1-SNAPSHOT.jar"]
92 $ echo "CI/CD job completed, APP image should be available at:"
93 CI/CD job completed, APP image should be available at:
94 $ echo ${CI_REGISTRY_IMAGE}:${CI_REGISTRY_TAG_DEV}
95 harbor.corp.local/daniel_project1/docker-sample:ci-cd-v3
96 Running after_script
97 Saving cache
98 Uploading artifacts for successful job
99 Job succeeded

```

- If Pipeline execution is successful, in the target Harbor project/repository there should be a new built/pushed image(s) from GitLab project, tagged according to passed value of `CI_REGISTRY_TAG` variable and scanned for vulnerabilities, per project settings:

daniel_project1/docker-sample

Info Images

SCAN

COPY DIGEST

+ ADD LABELS

RETAG

X DELETE

<input type="checkbox"/>	Tag	Size	Pull Command	Vulnerabilities	Signed	Author	Creation Time	Docker Version
<input type="checkbox"/>	ci-cd-v1	80.53MB		<div><div>H</div><div>• 11 Total • 11 Fixable</div></div>			5/21/20, 12:18 PM	1.12.6
<input type="checkbox"/>	latest	80.53MB		<div><div>H</div><div>• 11 Total • 11 Fixable</div></div>			5/19/20, 12:47 PM	1.12.6

- Notes on additional GitLab CI/CD resources and Best practices:
 - Examples of end-to-end Docker container build and deployment automation via GitLab CI/CD pipelines can be found in various blogs such as:
<https://sanderknape.com/2019/02/automated-deployments-kubernetes-gitlab/>
 - Operations teams that just need to run stabilized builds may use “Auto DevOps” GitLab mode and run pipelines defined in **.gitlab-ci.yml** as in example above.

- Developers may need to create their own customized pipelines, please see GitLab documentation: <https://gitlab.acelab.local/help/ci/pipelines/settings#custom-ci-configuration-path>

6. Configure and Execute CI/CD Pipeline for Deployment of Application from Registry to K8s Cluster

In this section we will continue building our end-to-end CI/CD pipeline that implements automated deployment of container images from Harbor registry (pushed by previous CI/CD stages) into TKGI Kubernetes cluster integrated with our project

- Similar to how it was done in Section 3 (for deployment of GitLab Runner into the **mgr-cluster-test1** K8s cluster), we will integrate a TKGI K8s cluster **mgr-cluster-test2** for deployment of an application to our GitLab project. Below are properties of an integrated K8s cluster for our project “dockersample”:

See and edit the details for your Kubernetes cluster

Kubernetes cluster name

mgr-cluster-test2

API URL

<https://mgr-cluster-test2.corp.local:8443>

CA Certificate

```
-----BEGIN CERTIFICATE-----
MIIC+zCAAgAwIBAgIUaK18BTnT0eLgNDGgEfeQVUch2n4wDQYJKoZIhvcNAQEL
BQAwDTELMAkGA1UEAxMCY2EwHhcNMjAwNDIwMDIwMDIwMDIwMDIwMDIwMDIwMDIw
WjANMQswCQYDVQQDEwJyTCCASlwdQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEB
ALQ7G48MxgOg95z0ci2ZBkJRWogkmpQNOQrIox2PvI0F/CJmIDcMAK2OGpQASDk6Y
dYMAAt3C30x7YvcmKTzLQ5mkmPU2fPGnXm5W0SiefnGoQbPAQJ9EuqXGHykKh22
a47xRElqJdqficsEZ4Ulcgb/QackKljG6p+MvPalb3rGQHfCvcjhyynyEO+pyo9
8xTwkk8P0V51V+sBUnc9/drB80pHalnOPIRVZoUEZesjQNEsXS9BK2sqZXHrBjd5
fuoeMzrW1pkjXqGBhpy+QPHYs8K1ozeNjz9NMopCwo4JwQ9fE6/4Jl1hO5diouv
hm6t9Lek83rz3ygcceJ0/aM0CAwEAATMFewHQYDVR0OBByEFFVJ/lj7AcGqLS71
```

Service Token

..... Show

☒ **RBAC-enabled cluster**
Enable this setting if using role-based access control (RBAC). This option will allow you to install applications on RBAC clusters.

☐ **GitLab-managed cluster**
Allow GitLab to manage namespace and service accounts for this cluster. [More information](#)

Project namespace (optional, unique)

qrcoe






The namespace associated with your project. This will be used for deploy boards, logs, and Web terminals.

Save changes

NOTE: Optional project namespace property will map into K8s namespace that would be set as a target namespace for applications deployment. In our example, that namespace

is “qrcode” and it will need to exist in the target K8s cluster when application will be deployed by CI/CD pipeline

- We will add 2 Kubernetes related files to our GitLab project repository:
 - Deployment descriptor for GitLab Service Account used to create service account for integration with K8s cluster above (gitlab-service-account.yaml)
 - Deployment descriptor for deployment of application containers into K8s cluster (deployment_harbor.yaml)

Name	Last commit	Last update
qrcode	Initial seed from cloned GitHub repo to test CI/CD pipeline	1 week ago
 .gitlab-ci.yml	Update .gitlab-ci.yml development uses CI_REGISTRY_TAG_DEV ta...	14 seconds ago
 README.md	Initial seed from cloned GitHub repo to test CI/CD pipeline	1 week ago
 deployment_harbor.yaml	Update deployment_harbor.yaml removed liveness and readiness ...	48 minutes ago
 gitlab-ci-docker.yaml	Initial seed from cloned GitHub repo to test CI/CD pipeline	1 week ago
 gitlab_service_account.yaml	Adding gitlab_service_account.yaml (already applied) and deploy...	3 hours ago

- Below is the content of **deployment_harbor.yaml** file – a pretty standard K8s Deployment descriptor that specifies image name, labels and number of replicas to be created in ReplicaSet as well as K8s secret with Harbor login credentials that would need to exist in the namespace where application will be deployed:

```

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: qrcode-java
  namespace: hello-world
  labels:
    app: qrcode
spec:
  replicas: 2
  selector:
    matchLabels:
      app: qrcode
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 33%
  template:
    metadata:
      labels:
        app: qrcode
    spec:

```

```

imagePullSecrets:
  - name: regcred
containers:
  - name: qrcode
    #NOTE: cicd-v3 tag is for deployemnt branch, will
    have to use another one for production
    image:
harbor.corp.local/daniel_project1/dockersample:cicd-v3
    ports:
      - containerPort: 8080

```

NOTE: in this version, the images tagged as **harbor.corp.local/daniel_project1/dockersample:cicd-v3** is expected to be present in the Harbor repository at the time of deployment to K8s. That tag value is contained in the CA_REGISTRY_TAG_DEV variable below

Variables ?

Environment variables are applied to environments via the runner. They can be protected by only exposing them to protected branches or tags. Additionally, they can be masked so they are hidden in job logs, though they must match certain regexp requirements to do so. You can use environment variables for passwords, secret keys, or whatever you want. You may also add variables that are made available to the running application by prepending the variable key with `K8S_SECRET_`. [More information](#)

Type	Key	Value	Protected	Masked	Environments	
Var	CI_REGISTRY	*****	X	X	All (default)	
Var	CI_REGISTRY_CA_CERT	*****	X	X	All (default)	
Var	CI_REGISTRY_IMAGE	*****	X	X	All (default)	
Var	CI_REGISTRY_PASSWORD	*****	X	X	All (default)	
Var	CI_REGISTRY_TAG_DEV	*****	X	X	All (default)	
Var	CI_REGISTRY_TAG_PROD	*****	X	X	All (default)	
Var	CI_REGISTRY_USER	*****	X	X	All (default)	

Reveal values Add Variable

- Now we need to add the stage to our CI/CD pipeline script (.gitlab-ci.yml defined in the Section 4) that will take care of establishing connection to target K8s cluster and performing all necessary steps to deploy our application using deployment descriptor above. For now, we only want to deploy the application only when pipeline on **master** branch is run hence the 'only ...' condition

```

...
deploy to development:
  stage: deploy
  image: "registry.gitlab.com/gitlab-org/cluster-
integration/auto-deploy-image:v0.15.0"

```

```

script:
  - echo "Runner in Environment:"
  - echo $CI_ENVIRONMENT_SLUG
  - echo "WILL USE BUILT IN K8s VARS PER ENVIRONENT:"
  - echo "KUBE_URL:"
  - echo $KUBE_URL
  - echo "KUBE_NAMESPACE:"
  - echo $KUBE_NAMESPACE
  - echo "Path to kubeconfig:"
  - echo $KUBECONFIG
  - echo "=====
  - echo "Trying to 'get nodes' using default
kubeconfig setting..."
  # will use syntax like kubectl config --
kubeconfig=config-demo set-cluster
  - kubectl --kubeconfig="${KUBECONFIG}" get nodes
  - echo "=====
  - echo "CHECK K8s current context set to:"
  - kubectl config current-context
  - echo "Getting Nodes info w/o using context:"
  - kubectl get nodes
  #- echo "-----
-----"
  # TODO: add error handling in case if NS exists
  - kubectl create ns $KUBE_NAMESPACE
  - echo "Creating regcred Docker login secret
'regcred'.."
  - kubectl create secret docker-registry regcred --
docker-server=$CI_REGISTRY --docker-
username=$CI_REGISTRY_USER --docker-
password=$CI_REGISTRY_PASSWORD --docker-
email=admin_harbor@acelcab.local
  - kubectl get secrets
  - echo "Will deploy QR Code container app to target
cluster/namespace:"
  - kubectl apply -f
$CI_PROJECT_DIR/deployment_harbor.yaml
  - echo "QR Code container app deployed to K8s
cluster, checking Pod status:"
  #- kubectl get po -n default
  - kubectl get po -n $KUBE_NAMESPACE
environment:
  name: development
  #deploy app to K8s only when a pipeline on master
branch is run
only:
  - master

```

NOTES:

- We are using a different container image to run jobs for **deploy** stage, "registry.gitlab.com/gitlab-org/cluster-integration/auto-deploy-image:v0.15.0" that has kubectl CLI utility pre-installed
 - \$KUBE_URL, \$KUBE_TOKEN, \$KUBE_NAMESPACE and \$KUBECONFIG are K8s environment variables that become available to CI/CD script once an **environment** is defined in there, as in the last 2 lines above (see details in documentation: <https://docs.gitlab.com/ee/user/project/clusters/#deploying-to-a-kubernetes-cluster>)
 - \$KUBECONFIG variable contains path to K8s configuration file placed on the 'builder' image that can be used by 'kubectl' CLI utility, pretty much the only parameter needed for access to target cluster
 - Namespace contained in the \$KUBE_NAMESPACE needs to exist in the target K8s cluster, so it is getting created by `kubectl create ns $KUBE_NAMESPACE` command
 - A secret containing Harbor private registry credentials for deployment of an image needs to exist in that namespace and gets created by `kubectl create secret docker-registry regcred ...` command.
 - Actual application deployment is done by running `kubectl apply -f $CI_PROJECT_DIR/deployment_harbor.yaml` command that is using deployment descriptor above
 - As mentioned above, our pipeline deploy the application into **development** environment only when pipeline on **master** branch is run
 - Multiple '- echo ' operators are placed into the script for debugging purposes and can be removed
- When our project CI/CD pipeline is executed end-to-end, it first runs a unit test stage (not implemented, has a placeholder), then rebuilds a Docker image from the source code and instructions contained in the Dockerfile, tags it with value \$CI_REGISTRY_TAG_DEV and pushes it into corresponding repository in Harbor.
 - An output of **docker build package** script section is shown below:

```

65 $ /bin/sh -c 'cat /dev/urandom | tr -dc 'a-z0-9' | fold -w 64 | xargs printf '%s\n' | shasum -a 256 | sed 's/ /,/' | tr -d '\n' | xargs echo'
66 ready 19.67s-30.12s166 17 aws_credentials.go:72] while getting AWS credentials: NoCredentialProviders: no valid providers in chain - Deprecated
67 For verbose messaging see aws.Config.CredentialsChainVerboseErrors
68 INFO[0041] Retrieving image manifest java:8-jdk-alpine
69 INFO[0044] Retrieving image manifest java:8-jdk-alpine
70 INFO[0047] Built cross stage deps: map[]
71 INFO[0047] Retrieving image manifest java:8-jdk-alpine
72 INFO[0047] Retrieving image manifest java:8-jdk-alpine
73 INFO[0050] Executing 0 build triggers
74 INFO[0050] Unpacking rootfs as cmd COPY ./target/qrcode-0.0.1-SNAPSHOT.jar /usr/app/ requires it.
75 INFO[0077] COPY ./target/qrcode-0.0.1-SNAPSHOT.jar /usr/app/
76 INFO[0079] Resolving 1 paths
77 INFO[0079] Taking snapshot of files...
78 INFO[0084] WORKDIR /usr/app
79 INFO[0084] cmd: workdir
80 INFO[0084] Changed working directory to /usr/app
81 INFO[0084] RUN sh -c 'touch qrcode-0.0.1-SNAPSHOT.jar'
82 INFO[0084] Taking snapshot of full filesystem...
83 INFO[0084] Resolving 1672 paths
84 INFO[0088] cmd: /bin/sh
85 INFO[0088] args: [-c sh -c 'touch qrcode-0.0.1-SNAPSHOT.jar']
86 INFO[0088] Running: [/bin/sh -c sh -c 'touch qrcode-0.0.1-SNAPSHOT.jar']
87 INFO[0088] Taking snapshot of full filesystem...
88 INFO[0088] Resolving 1672 paths
89 INFO[0090] ENTRYPOINT ["java", "-jar", "qrcode-0.0.1-SNAPSHOT.jar"]
90 $ echo "CI/CD job completed, APP image should be available at:"
91 CI/CD job completed, APP image should be available at:
92 $ echo ${CI_REGISTRY_IMAGE}:${CI_REGISTRY_TAG_DEV}
93 harbor.corp.local/daniel_project1/docker-sample:ci-cd-v3
94 Running after_script
95 Saving cache
96 Uploading artifacts for successful job
97 Job succeeded

```

Then script in the **deploy** stage performs deployment of checked container image from Harbor registry according to deployment descriptor into designated namespace in the integrated K8s cluster:

```

59 CHECK K8s current context set to:
60 $ kubectl config current-context
61 gitlab-deploy
62 $ echo "Getting Nodes info w/o using context:"
63 Getting Nodes info w/o using context:
64 $ kubectl get nodes
65 NAME STATUS ROLES AGE VERSION
66 5ab2ae3a-2315-42f1-bc30-b32ae3879059 Ready <none> 20d v1.16.7+vmware.1
67 e45756cc-114c-4918-b4ba-55268f4e7258 Ready <none> 18d v1.16.7+vmware.1
68 $ kubectl create ns $K8NS_NAMESPACE
69 namespace/qrcode created
70 $ echo "Creating regcred Docker login secret 'regcred'.."
71 Creating regcred Docker login secret 'regcred'..
72 $ kubectl create secret docker-registry regcred --docker-server=${CI_REGISTRY} --docker-username=${CI_REGISTRY_USER} --docker-password=${CI_REGISTRY_PASSWORD} --docker-email=ad
73 min.harbor@acelcab.local
74 secret/regcred created
75 $ kubectl get secrets
76 NAME TYPE DATA AGE
77 regcred kubernetes.io/dockerconfigjson 1 1s
78 $ echo "Will deploy QR Code container app to target cluster/namespace:"
79 Will deploy QR Code container app to target cluster/namespace:
80 $ kubectl apply -f ${CI_PROJECT_DIR}/deployment_harbor.yaml
81 deployment.apps/qrcode-java created
82 $ echo "QR Code container app deployed to K8s cluster, checking Pod status:"
83 QR Code container app deployed to K8s cluster, checking Pod status:
84 $ kubectl get po -n $K8NS_NAMESPACE
85 NAME READY STATUS RESTARTS AGE
86 qrcode-java-68587f4bcd-2vsw 0/1 ContainerCreating 0 1s
87 qrcode-java-68587f4bcd-qtrx7 0/1 ContainerCreating 0 1s
88 Running after_script
89 Saving cache
90 Uploading artifacts for successful job
91 Job succeeded

```

And checking directly in the **mgr-cluster-test2** K8s cluster, we can see 2 replicas of **qrcode-java** running, as expected:

```
kubectl get po -n qrcode
```

NAME	READY	STATUS	RESTARTS
AGE			

```

qrcline-java-68587f4bcd-2vsbw    1/1      Running    0
17m
qrcline-java-68587f4bcd-qtr7    1/1      Running    0
17m

```

and Pod details

```

$ kubectl describe po qrcline-java-68587f4bcd-2vsbw -n qrcline
Name:          qrcline-java-68587f4bcd-2vsbw
Namespace:     qrcline
Priority:       0
Node:          e45756cc-114c-4918-b4ba-55268f4e7250/172.15.2.4
Start Time:    Mon, 01 Jun 2020 18:59:16 +0000
Labels:        app=qrcline
               pod-template-hash=68587f4bcd
Annotations:   <none>
Status:        Running
IP:            172.16.26.2
IPs:
  IP:          172.16.26.2
Controlled By: ReplicaSet/qrcline-java-68587f4bcd
Containers:
  qrcline:
    Container ID:
docker://233c8775a01d4ae08bc54f6496fea435f9cea78e5e3d3ec2472c7
4e9cd1b5f4d
    Image:
harbor.corp.local/daniel_project1/dockersample:cicd-v3
    Image ID:
docker-
pullable://harbor.corp.local/daniel_project1/dockersample@sha2
56:2cd6af4371930ce694919098858cfed5cb0c3d4261c1365e9d4c11fc31c
fa387
    Port:      8080/TCP
    Host Port: 0/TCP
    State:     Running
      Started: Mon, 01 Jun 2020 18:59:19 +0000
    Ready:     True
    Restart Count: 0
    Environment: <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from
default-token-c6f45 (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True

```

Finally, using kube-proxy port forwarding we can post a request to one of the Pods and get a response:

```
curl -d '{"merchantID":"123", "merchantName":"daniel"}' -H
"Content-Type: application/json" -X POST
http://localhost:8080/generateQRCode --output QRCode.out
  % Total    % Received % Xferd  Average Speed   Time    Time
Time  Current                      Dload  Upload   Total   Spent
Left  Speed
100  312  100    267  100    45   2321    391 --:--:-- --:--:-- -
-:--:-- 2713
```

NOTES:

- In addition to **development** environment, other environments (e.g. **staging**, **production etc.**) can be defined in the CI/CD script sections and contain different deployment instructions/parameters such as Docker image tags. That's the purpose of having CI_REGISTRY_TAG_PROD variable. Please see documentation for CI/CD pipelines: <https://docs.gitlab.com/ee/ci/pipelines/index.html>
- There may be **test** stage defined, with jobs for testing deployed application.
- As in our example, GitLab Runner containers and deployed Applications may run in different K8s clusters (in fact, that may be preferred) - they just need to have networking/DNS access to each other as **mgr-cluster-test1** and **mgr-cluster-test2** do.