An IBM Document from

IBM Z DevOps Acceleration Program

September, 2020 - Release 1.4

Integrating IBM z/OS platform in CI pipelines with Gitlab

Mathieu Dalbin
mathieu.dalbin@fr.ibm.com

Shabbir Moledina
shabbirm@ca.ibm.com

Abstract
Install and configure Gitlab runners for z/OS
Setup CI pipelines for z/OS using Gitlab CI

Table of content

1 Introduction

Gitlab is a popular solution for Source Code Management but it also includes several features in the DevOps space to extend its scope of activities and enhance user experience. One of these features is the Continuous Integration (CI) mechanism, which provides orchestration capabilities and brings automation closer to source repositories.

The main object in Gitlab is a project which combines a Git repository with project-related features like CI: with this concept, there is tight integration between the project and its automation, linking a repository with its CI pipeline. Thus, the CI pipeline is defined in the project as a YAML file stored in the repository, and it is usually called "*.gitlab-ci.yml*".

The CI feature relies on a server-runners architecture: the Gitlab server owns the pipeline definitions for each project and delegates its work to agents installed on target environments. In Gitlab terminology, agents are called runners and can be customized to perform specific actions depending on the type of artifacts they are working with. The most popular executors are Docker and Shell, but other types of executors can be used to support specific configurations. Documentation related to the Gitlab runners can be found at https://docs.gitlab.com/runner/.

Runners are registered in the Gitlab server configuration and can be shared or dedicated to a project, depending on users' needs. When a pipeline is triggered for execution, the Gitlab server parses the pipeline definition file and pipeline actions are sent to an assigned runner for execution.

2 Integrating IBM z/OS platform into Gitlab CI

Gitlab runners are written in Go, a language that is not yet supported by IBM z/OS (at the time of writing), so they cannot run natively in this environment. To circumvent this situation, the best option is to use a specific type of executor, which leverages the SSH protocol.

The SSH runners are executed on a supported platform (Windows, Linux or MacOS) and connect to a target machine through SSH for the pipeline execution. In that configuration, SSH runners act like gateways to connect platforms: the Gitlab server will send the pipeline actions to the SSH runner which will forward them to the target machine, in this case the z/OS environment. The different stages of the pipeline will be executed on the target z/OS machine and results will be sent back to the Gitlab server through the same mechanism.

With the 13.2 version released in July 2020, Gitlab supports the execution of runners on Linux on z machines and proposes a docker image of the runner for this platform. This support extends the capabilities of Gitlab CI on Mainframe, allowing the full Gitlab CI stack to be deployed on the Linux on z environment: the use of SSH executors, as described in this document, is applicable to Linux on z runners.

The docker image can also run under the control of z/OS Container Extension (zCX) which is available with z/OS 2.4. In this configuration, the necessary components for the integration of z/OS with Gitlab CI run in a single z/OS environment. The configuration of Linux on z Gitlab runners is not in scope for this document, but information and guidelines on setting up zCX can be found in the “Getting started with z/OS Container Extensions and Docker” IBM RedBook (SG24-8457-00) at https://www.redbooks.ibm.com/abstracts/sg248457.html?Open.

2.1 Pre-requisites

As Gitlab CI runners only support the Bash shell environment, the default shell provided by z/OS is insufficient to support this configuration. As a pre-requisite, the Bash shell must be installed on z/OS. This Bash shell is provided by Rocket, as part of their Open Source Languages and Tools for z/OS offering (https://www.rocketsoftware.com/zos-open-source/tools).

As the execution of the pipeline is performed through an SSH channel, it is important to have the Bash shell available for the user who will execute the build actions. More importantly, the Bash shell must be set to be the first program executed by the user when logging into z/OS Unix System Services. This parameter is controlled through the PROGRAM keyword in the RACF’s user definition, as part of the OMVS segment. To change the PROGRAM value for an existing user, please customize and use the following RACF ALTUSER command:

```
ALU #USERID# OMVS(Program(#PATH_TO_BASH#))
```

Where **#USERID#** is the User ID to be updated and **#PATH_TO_BASH#** is set to the path of the Bash shell executable (do not surround with quotes).

During the execution of a pipeline on the target machine, the script generated by Gitlab CI is looking for the bash executable at a specific location, in /bin/bash. As bash is not standard on z/OS, this location doesn’t exist, and a symbolic link must be created to point to the actual path of the bash executable. Please customize and use the following command to create a symbolic link to point to right location:

```
ln -s #PATH_TO_BASH# /bin/bash
```

The Git for z/OS client must also be installed and configured on the z/OS USS platform. It is available through IBM Passport Advantage website (https://www.ibm.com/software/passportadvantage/) or it can be obtained, as for Bash, from Rocket's Open Source Languages and Tools for z/OS packaging (https://www.rocketsoftware.com/zos-open-source/tools). Installation guidance can be found at https://www.ibm.com/support/knowledgecenter/en/SS6T76_1.0.9/setup_git_on_uss.html.

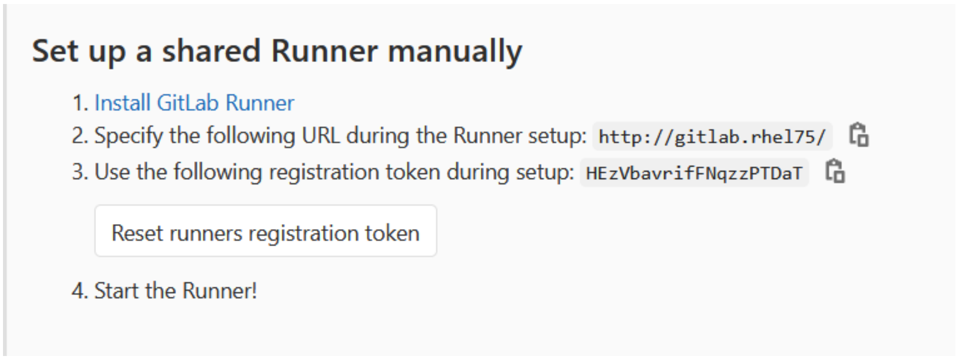
Git for z/OS is a free software with no support, but IBM support can be provided through the "IBM Elite Support for Rocket Git for zOS" offering (https://www-01.ibm.com/common/ssi/ShowDoc.wss?docURL=/common/ssi/rep_ca/2/897/ENUS219-032/index.html&request_locale=en).

2.2 Installing the Gitlab runner

First, download the gitlab-runner distribution for the appropriate platform at https://docs.gitlab.com/runner/install/. In this configuration, the SSH Gitlab runner is installed on a Linux x86 Red Hat distribution, but similar installation and configuration can be performed on other supported platforms.

In Linux, install the downloaded package with the command *rpm -i gitlab-runner_amd64.rpm*.

Then register the runner to the Gitlab server, by issuing the *sudo gitlab-runner register* command. The configurator prompts for several pieces of information that are displayed in the Gitlab server Administration area (Runners section):



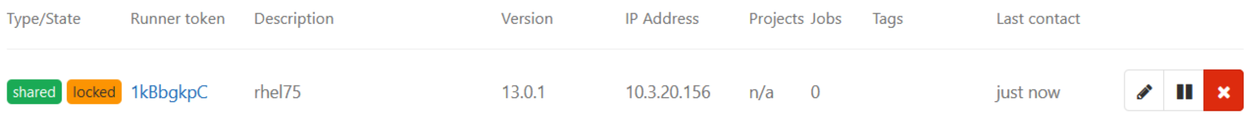
Set up a shared Runner manually

1. Install GitLab Runner
2. Specify the following URL during the Runner setup: `http://gitlab.rhel75/`
3. Use the following registration token during setup: `HEzVbavrifFNqzzPTDaT`

4. Start the Runner!

Provide the Gitlab server URL and the registration token to register the runner. Then provide a description for this runner and leave empty when prompted for tags. Provide the type of executor by typing *ssh* and provide the necessary information for SSH communication (IP or hostname of the target z/OS machine, port, username and password or path to the SSH identity file). When finished, start the runner by issuing the *gitlab-runner start* command.

The runner should appear in the Runners section of the Gitlab Administration area:



Type/State	Runner token	Description	Version	IP Address	Projects	Jobs	Tags	Last contact
shared locked	1kBbgkpC	rhel75	13.0.1	10.3.20.156	n/a	0		just now

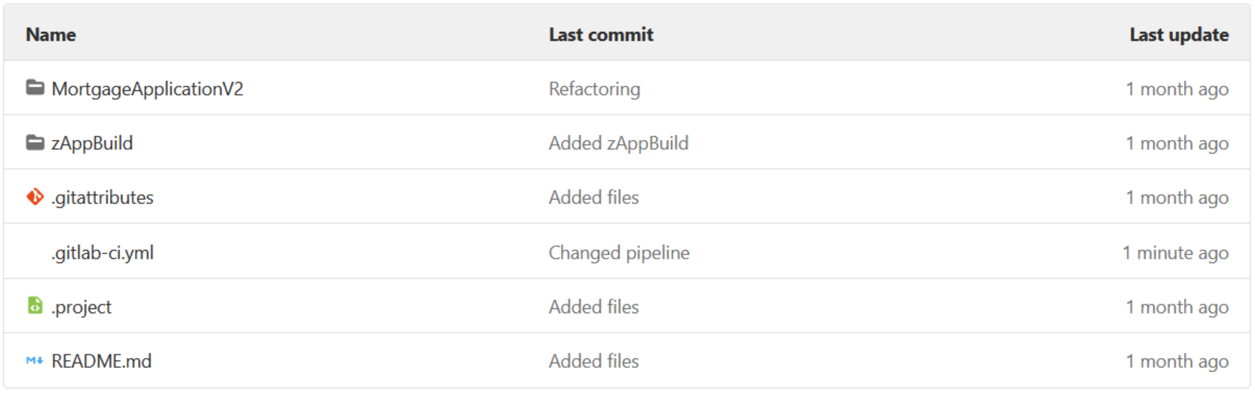
3 Setting up pipelines in Gitlab CI

Once the Gitlab SSH runner is set up for z/OS, CI pipelines can be executed on Mainframe. The definition of pipelines in Gitlab CI are done through a configuration file, called ".gitlab-ci.yml" and stored at the root level of the project's repository. The pipeline definition reference is available at https://docs.gitlab.com/ee/ci/yaml/README.html

The purpose of this document is not to provide a complete pipeline sample, instead it is meant to show an implementation of pipelines using available features in Gitlab CI, to build a Mainframe application on z/OS. Compilation and linkage editing of Mainframe source files will be performed with the Dependency-Based Build solution (DBB) and the zAppBuild framework, which facilitates the orchestration of build operations.

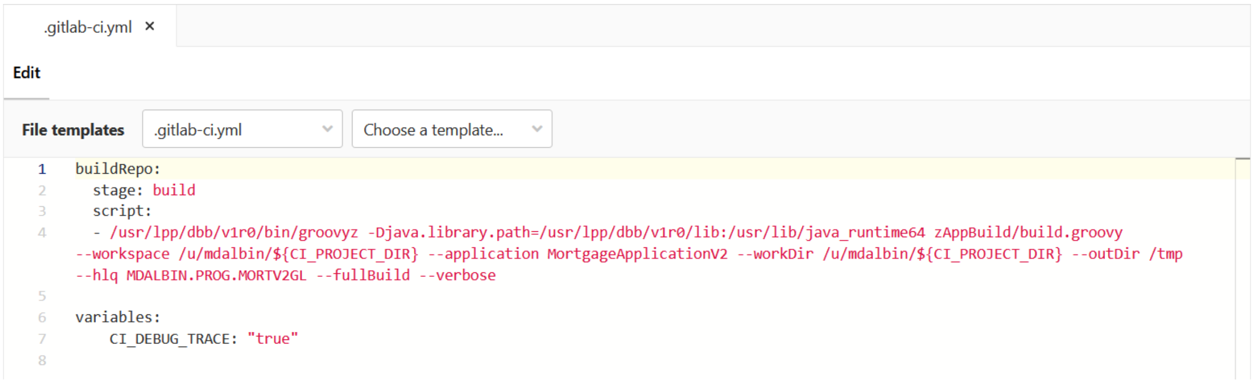
3.1 Building a z/OS application with a simple pipeline

In this first use case, the project contains all the necessary materials to build the application: zAppBuild is part of the project and has been customized to meet the environment's requirements. This scenario is using the sample Mortgage application, as available in GitHub at https://github.com/IBM/dbb-zappbuild.



Name	Last commit	Last update
MortgageApplicationV2	Refactoring	1 month ago
zAppBuild	Added zAppBuild	1 month ago
.gitattributes	Added files	1 month ago
.gitlab-ci.yml	Changed pipeline	1 minute ago
.project	Added files	1 month ago
README.md	Added files	1 month ago

The pipeline is made up of a single operation, which is the build of the full z/OS Cobol application with zAppBuild and DBB. It only contains one step defined as part of the build stage:



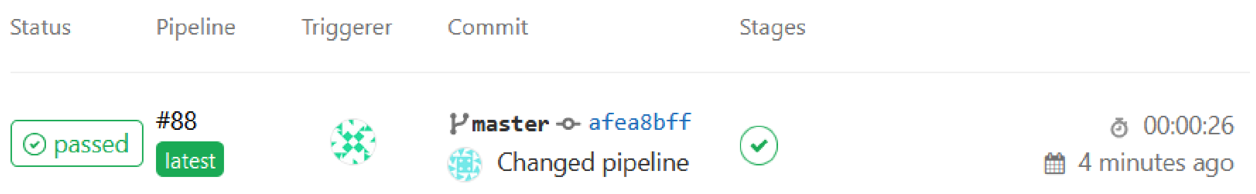
```
buildRepo:
  stage: build
  script:
  - /usr/lpp/dbb/v1r0/bin/groovyz -Djava.library.path=/usr/lpp/dbb/v1r0/lib:/usr/lib/java_runtime64 zAppBuild/build.groovy
--workspace /u/mdalbin/${CI_PROJECT_DIR} --application MortgageApplicationV2 --workDir /u/mdalbin/${CI_PROJECT_DIR} --outDir /tmp
--hlq MDALBIN.PROG.MORTV2GL --fullBuild --verbose

variables:
    CI_DEBUG_TRACE: "true"

```

Text version of pipeline:

```
buildRepo:
 stage: build
 script:
 - /usr/lpp/dbb/v1r0/bin/groovyz -Djava.library.path=/usr/lpp/dbb/v1r0/lib:/usr/lib/java_runtime64
zAppBuild/build.groovy --workspace /u/mdalbin/${CI_PROJECT_DIR} --application
MortgageApplicationV2 --workDir /u/mdalbin/${CI_PROJECT_DIR} --outDir /tmp --hlq
MDALBIN.PROG.MORTV2GL --fullBuild --verbose

variables:
   CI_DEBUG_TRACE: "true"
```

For convenience, the extended trace has been activated to facilitate debugging, but it is recommended to remove it when the pipeline performs the desired actions correctly.

In the Gitlab CI section of this project, the execution of this pipeline was manually triggered and finished successfully in 26 seconds:



Status	Pipeline	Triggerer	Commit	Stages	
passed	#88 latest		master afea8bff Changed pipeline	✓	00:00:26 4 minutes ago

When drilling down and checking the output log of the buildRepo job, the execution of the DBB command showed that the full build of the application was performed, building 8 Cobol files successfully.

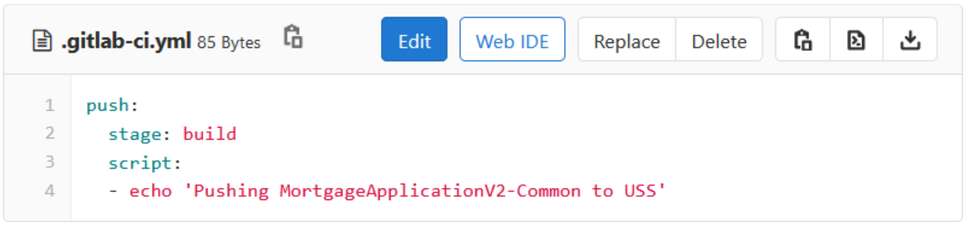
3.2 Building a z/OS application with multiple projects

Mainframe applications are usually communicating with each other, through interfaces defined in copybooks. In order to be built correctly, an application must include the copybooks of the other applications it communicates with. With Git, this generally implies that the application's main repository is insufficient for building, and other repositories that contain shared copybooks from other applications are needed for a successful build. In Gitlab CI, this situation implies referencing other projects, which contain sources of other applications.

This situation can be managed with multi-project pipelines, a feature of Gitlab CI which permits the triggering of pipelines in other projects. Reference for this feature is available at https://docs.gitlab.com/ee/ci/multi_project_pipelines.html

In this use case, the Mortgage application has been split into 2 components, EPSC and EPSM, and a 3rd project contains common copybooks used by the two components. To be built, the EPSC component needs to reference copybooks owned by the EPSM project and common copybooks. A 4th project contains the zAppBuild framework, customized for the z/OS environment. The correct build of the EPSC component requires the 4 projects to be transferred to z/OS USS.

Using the multi-project pipelines feature, it is possible to trigger the execution of the external projects pipeline, which solves this complex situation. To be cloned on z/OS USS, the pipelines for the EPSM project, the Common project and the zAppBuild project will just contain a dummy operation, as cloning is automatically generated by the Gitlab server in the execution of the pipeline. The pipeline definition for the Common project is as follows:



```yaml
push:
  stage: build
  script:
  - echo 'Pushing MortgageApplicationV2-Common to USS'
```

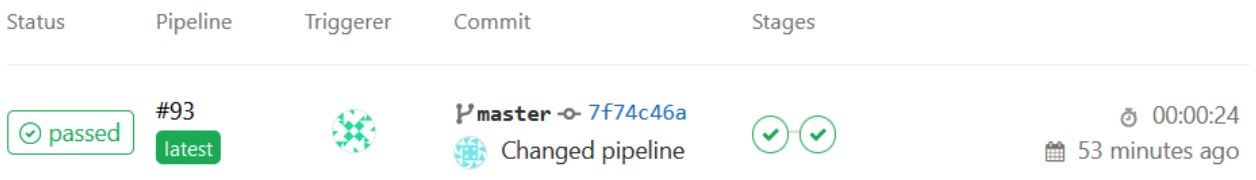
Similar pipeline definitions are configured for the EPSM project and for the zAppBuild project.

For the EPSC project, the pipeline contains steps in the ".pre" stage, which is the first stage executed in the pipeline. These steps trigger downstream pipelines for the EPSM, Common and zAppBuild projects.

```
zAppBuild-Update:
  stage: .pre
  trigger:
    project: root/zappbuildgitlab
    strategy: depend

MortgageApplicationV2-Common-Update:
  stage: .pre
  trigger:
    project: root/gl-mortgageapplicationv2-common
    strategy: depend

MortgageApplicationV2-EPSM-Update:
  stage: .pre
  trigger:
    project: root/gl-mortgageapplicationv2-epsm
    strategy: depend

buildRepo:
  stage: build
  script:
  - cd
/u/mdalbin/${CI_BUILDS_DIR}/${CI_RUNNER_SHORT_TOKEN}/${CI_CONCURRENT_ID}/root/zappbuildgit
lab/
  - /usr/lpp/dbb/v1r0/bin/groovyz -Djava.library.path=/usr/lpp/dbb/v1r0/lib:/usr/lib/java_runtime64
build.groovy --workspace
/u/mdalbin/${CI_BUILDS_DIR}/${CI_RUNNER_SHORT_TOKEN}/${CI_CONCURRENT_ID}/root/ --
application gl-mortgageapplicationv2-common --workDir /u/mdalbin/${CI_PROJECT_DIR} --outDir /tmp -
-hlq MDALBIN.PROG.MORTV2GL --fullBuild --verbose

stages:
  - .pre
  - build
```

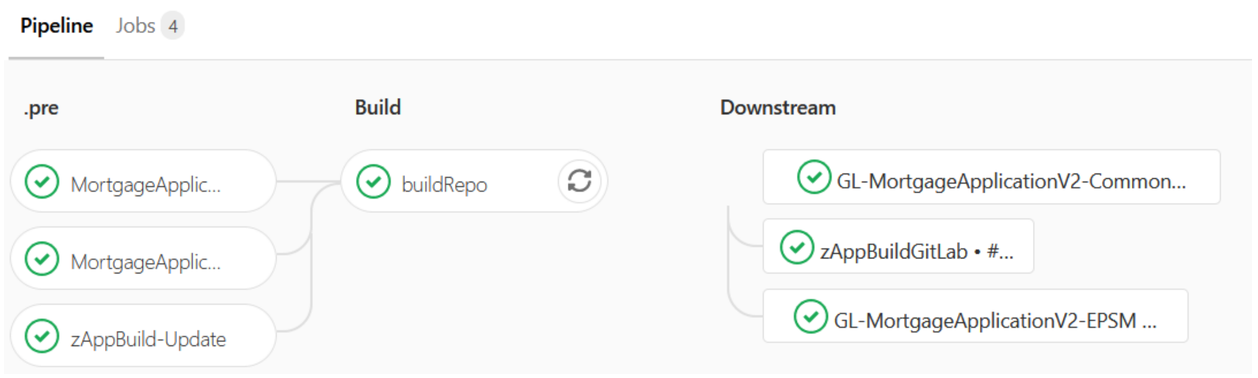
In this configuration, the jobs zAppBuild-Update, MortgageApplicationV2-Common-Update and MortgageApplicationV2-EPSM-Update are executing first, triggering downstream pipelines for the different projects. With the keyword "*strategy: depend*", the main pipeline will wait until all the jobs in the ".pre" stage are finished before moving to the build stage, which contains the buildRepo job that performs the build of the EPSC component with DBB.

In the Gitlab CI section of the EPSC project, the correct execution of the pipeline can be checked:



Status	Pipeline	Triggerer	Commit	Stages	
passed	#93 latest		master 7f74c46a Changed pipeline	✓ ✓	00:00:24 53 minutes ago

By selecting the pipeline, the actual sequence of downstream jobs is displayed:



Zooming on the buildRepo job, the correct execution of the build process for the EPSC component shows that 8 programs were successfully built by DBB.