April, 2023 - Release 1.7

An IBM Document from

IBM Z DevOps Acceleration Program

Integrating IBM z/OS platform in CI/CD pipelines with GitLab

Mathieu Dalbin
mathieu.dalbin@fr.ibm.com

Shabbir Moledina
shabbirm@ca.ibm.com

Abstract

Installation and configuration of GitLab runners for z/OS, pipelines setup using GitLab CI/CD features and extensions with IBM Z DevOps solutions

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/Continuous Deployment (CI/CD) 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/CD: with this concept, there is a tight integration between the project and its automation, linking a repository with its CI/CD pipeline. Thus, the CI/CD pipeline is defined in the project as a YAML file stored in the repository, and it is usually called *.gitlab-ci.yml*.

The CI/CD 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's 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.

The purpose of this document is to share real use cases on implementing a pipeline on z/OS, using multiple-project pipelines, GitLab CI/CD features and extended capabilities brought by various IBM Z DevOps solutions.

2 Integrating IBM z/OS platform into GitLab CI/CD

GitLab runners are written in Go, a modern language that is now available on IBM z/OS (this availability was announced in March 2021[1]). However, GitLab runners for the z/OS platform are not yet available (at the time of writing this document), 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-based GitLab runners in zCX 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)[2].

2.1 Pre-requisites

As GitLab CI/CD 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[3].

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).

[1] https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?infotype=AN&subtype=CA&htmlfid=897/ENUS221-044&appname=USN
[2] https://www.redbooks.ibm.com/abstracts/sg248457.html?Open
[3] https://www.rocketsoftware.com/zos-open-source/tools

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 consider customizing the z/OS environment by creating a symbolic link to point to right location:

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

As the */bin* folder belongs to a filesystem shipped by z/OS, it may be mounted as read-only, so the creation of the symbolic link fails. The */* filesystem needs to be mounted as read-write to allow the creation of the symbolic link. The following command can be used to mount the */* filesystem as read-write, just for the time of symbolic link creation:

```
chmount -w /
```

When mounted as read-write, the creation of the symbolic link should be successful. The */* filesystem should be mounted back as read-only after the creation of the link, using that command:

```
chmount -r /
```

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[4] or it can be obtained, as for Bash, from Rocket's Open Source Languages and Tools for z/OS packaging. Installation guidance can be found on the IBM Documentation page for DBB[5].

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[6].

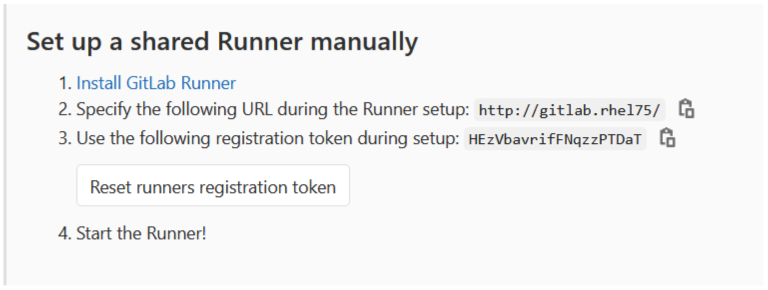
[4] https://www.ibm.com/software/passportadvantage/
[5] https://www.ibm.com/docs/en/dbb/1.1.0?topic=dbb-setting-up-git-uss
[6] https://www-01.ibm.com/common/ssi/ShowDoc.wss?docURL=/common/ssi/rep_ca/2/897/ENUS219-032/index.html&request_locale=en

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):



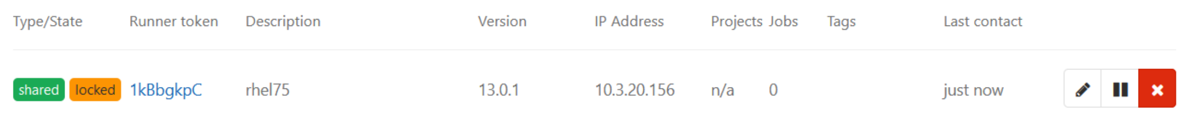
Set up a shared Runner manually

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 (tags can be modified later if necessary). 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 *sudo 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

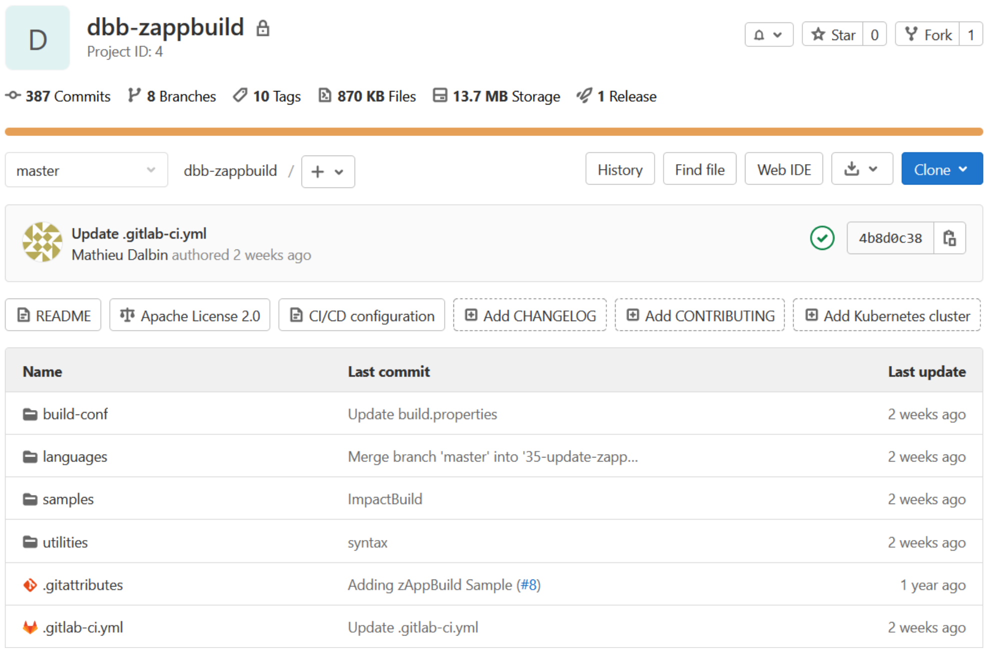
3 Setting up an initial pipeline with GitLab CI/CD

Once the GitLab SSH runner is set up for z/OS, CI/CD 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 be default at the root level of the project's repository (it can be changed in the project's CI/CD configuration). GitLab provides a complete documentation about keyworks that can be used in pipeline definitions[7].

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/CD, 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 *MortgageApplication* application, as available in GitHub at https://github.com/IBM/dbb-zappbuild. The *MortgageApplication* application is stored in the *samples* subfolder:



dbb-zappbuild
Project ID: 4

387 Commits · 8 Branches · 10 Tags · 870 KB Files · 13.7 MB Storage · 1 Release

master · dbb-zappbuild /

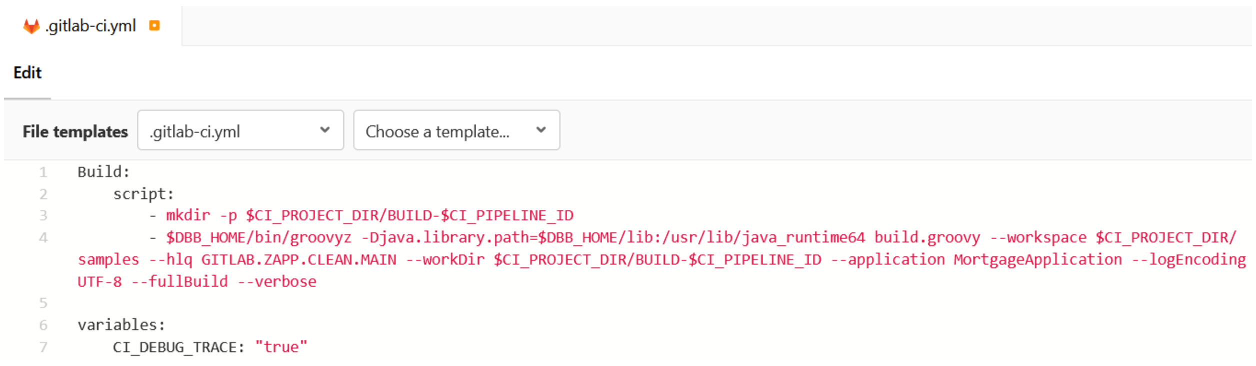
Update .gitlab-ci.yml
Mathieu Dalbin authored 2 weeks ago · 4b8d0c38

README · Apache License 2.0 · CI/CD configuration · Add CHANGELOG · Add CONTRIBUTING · Add Kubernetes cluster

Name	Last commit	Last update
build-conf	Update build.properties	2 weeks ago
languages	Merge branch 'master' into '35-update-zapp...	2 weeks ago
samples	ImpactBuild	2 weeks ago
utilities	syntax	2 weeks ago
.gitattributes	Adding zAppBuild Sample (#8)	1 year ago
.gitlab-ci.yml	Update .gitlab-ci.yml	2 weeks ago

[7] https://docs.gitlab.com/ee/ci/yaml/README.html

The pipeline initially contains 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:



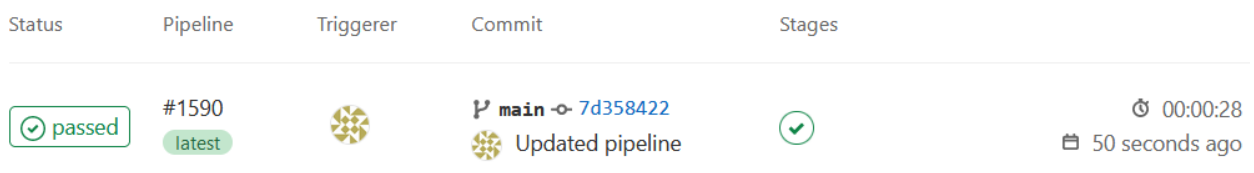
```
Build:
    script:
        - mkdir -p $CI_PROJECT_DIR/BUILD-$CI_PIPELINE_ID
        - $DBB_HOME/bin/groovyz -Djava.library.path=$DBB_HOME/lib:/usr/lib/java_runtime64 build.groovy --workspace $CI_PROJECT_DIR/
samples --hlq GITLAB.ZAPP.CLEAN.MAIN --workDir $CI_PROJECT_DIR/BUILD-$CI_PIPELINE_ID --application MortgageApplication --logEncoding
UTF-8 --fullBuild --verbose

variables:
    CI_DEBUG_TRACE: "true"
```

Text version of pipeline:

```
variables:
    DBB_HOME: "/usr/lpp/dbb/v1r0"

Build:
    script:
        - mkdir -p $CI_PROJECT_DIR/BUILD-$CI_PIPELINE_ID
        - $DBB_HOME/bin/groovyz -
Djava.library.path=$DBB_HOME/lib:/usr/lib/java_runtime64 build.groovy --
workspace $CI_PROJECT_DIR/samples --hlq GITLAB.ZAPP.CLEAN.MAIN --
workDir $CI_PROJECT_DIR/BUILD-$CI_PIPELINE_ID --application MortgageApplication --
logEncoding UTF-8 --fullBuild --verbose

variables:
    CI_DEBUG_TRACE: "true"
```

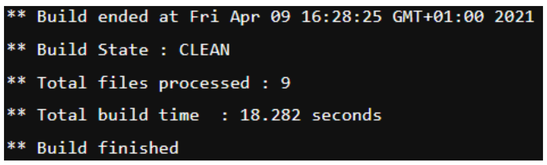
For convenience, the extended trace has been activated to facilitate debugging, but it is recommended to remove it when the pipeline performs the desired actions correctly. Please note the use of GitLab CI/CD variables which are available in the pipeline definition: in this case, the *$CI_PROJECT_DIR* variable contains the path where the project is checked out on the target machine, and the *$CI_PIPELINE_ID* variable contains the unique number associated with the pipeline execution. These variables are defined in the GitLab CI/CD documentation[8].

In the GitLab CI/CD section of this project, the execution of this pipeline was manually triggered and finished successfully in 28 seconds:



Status	Pipeline	Triggerer	Commit	Stages	
passed	#1590 latest		main 7d358422 Updated pipeline	✓	00:00:28 50 seconds ago

When drilling down and checking the output log of the Build job, the execution of the DBB command showed that the full build of the application was performed, building 9 files successfully.



```
** Build ended at Fri Apr 09 16:28:25 GMT+01:00 2021
** Build State : CLEAN
** Total files processed : 9
** Total build time  : 18.282 seconds
** Build finished
```

[8] https://docs.gitlab.com/ee/ci/variables/predefined_variables.html

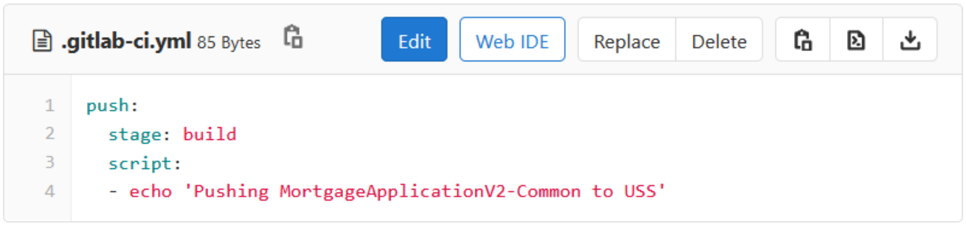
3.2 Building a z/OS application with multiple projects

Mainframe applications are usually exchanging data 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/CD, 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/CD which permits the triggering of pipelines in other projects. GitLab provides a reference page[9] for additional information.

In this use case, the Mortgage application has been split into 2 components, EPSC and EPSM, and a 3rd project contains the 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.

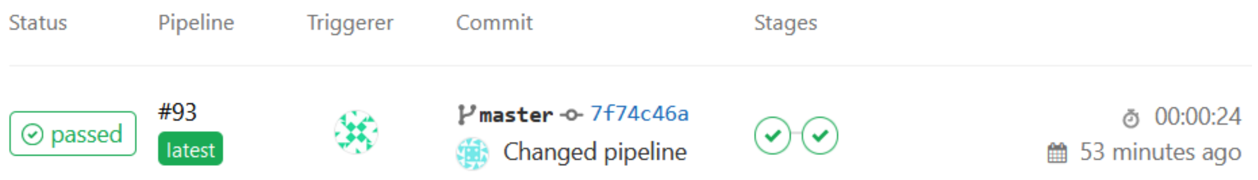
9 https://docs.gitlab.com/ee/ci/multi_project_pipelines.html

For the EPSC project, the pipeline contains steps in the *.pre* stage, which is configured to be 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}/${CI_CONCURRENT_ID}/root/zappbuildGitLab/
  - /usr/lpp/dbb/v1r0/bin/groovyz -
Djava.library.path=/usr/lpp/dbb/v1r0/lib:/usr/lib/java_runtime64 build.groovy --workspace
/u/mdalbin/${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
```

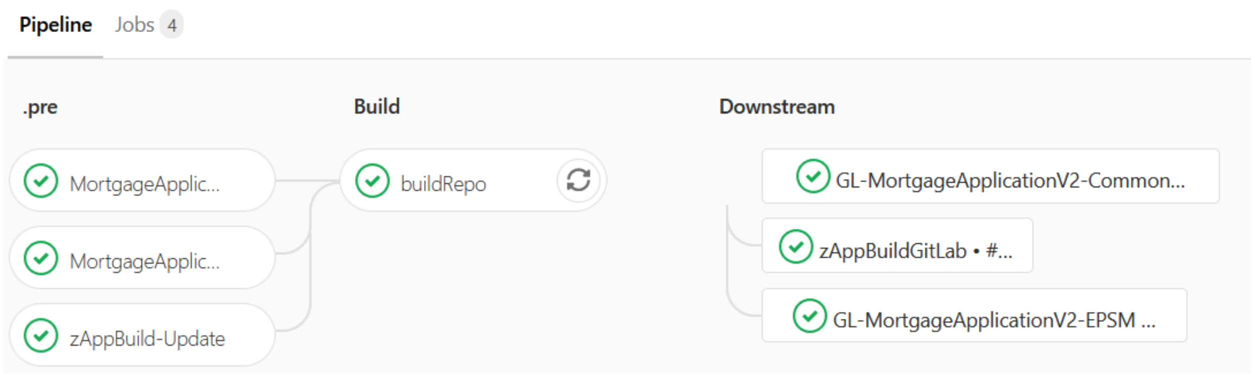
In this configuration, the jobs zAppBuild-Update, MortgageApplicationV2-Common-Update and MortgageApplicationV2-EPSM-Update are executing first, triggering downstream pipelines for the different projects. With the keyword *strategy: depend*, the main pipeline will wait until all the jobs in the *.pre* 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/CD 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.

More information on extending the scope of the build workspace can be found in the *Managing the build scope in IBM DBB builds with IBM zAppBuild* document[10].

[10] https://www.ibm.com/support/pages/node/6381788

4 Enriching the GitLab CI/CD pipeline with specific configurations

The GitLab CI/CD feature offers a lot of possibilities to control the execution of pipelines. The purpose of this section is not to list of all the possible configurations but to describe some options that provide benefits in the context of pipelines executing on the z/OS platform.

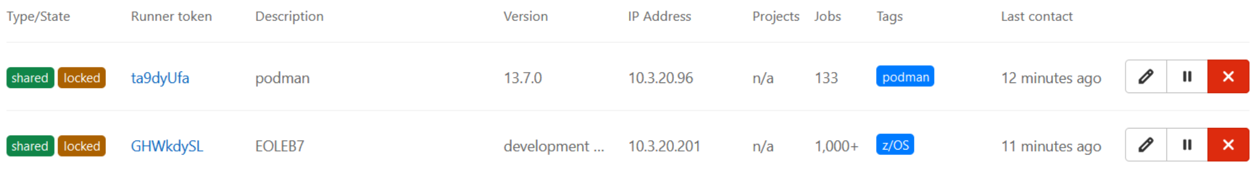
4.1 Using tags to control runners

In some situations, jobs must be executed on a targeted platform, which has a typical configuration or serves a specific purpose. In this case, it is important that the job to be executed is picked by the correct runner or the pipeline may end up in error.

To control by which runner the job can be picked and executed, tags can be set up at the pipeline level. For each job that has this type of restriction, a tag or a series of tags can be assigned to these jobs in their definition in the *.gitlab-ci.yml* file:

```
variables:
    DBB_HOME: "/usr/lpp/dbb/v1r0"
    ZAPPBUILD: $CI_BUILDS_DIR/$CI_RUNNER_SHORT_TOKEN/$CI_CONCURRENT_ID/dat/dbb-zappbuild

GenApp-Build:
    stage: Build
    tags: [z/OS]
    script:
        - $DBB_HOME/bin/groovyz -Djava.library.path=$DBB_HOME/lib:/usr/lib/java_runtime64
${ZAPPBUILD}/build.groovy --workspace $CI_PROJECT_DIR --workDir $CI_PROJECT_DIR/BUILD-
$CI_PIPELINE_ID --hlq GITLAB.GENAPP.DEMO --logEncoding UTF-8 --application genapp --verbose --
fullBuild  -cc
```

A requisite is to have tags set up at the runner level. When a runner is registered against a GitLab server instance, one or more tags can be specified, but it is also possible to modify these tags later, in the Admin Area of the GitLab server instance. For instance, a runner with the *z/OS* tag has been set up and registered as a shared runner (which means it can be used by any project).



Type/State	Runner token	Description	Version	IP Address	Projects	Jobs	Tags	Last contact
shared locked	ta9dyUfa	podman	13.7.0	10.3.20.96	n/a	133	podman	12 minutes ago
shared locked	GHWkdySL	EOLEB7	development ...	10.3.20.201	n/a	1,000+	z/OS	11 minutes ago

To ensure that a job requires to run on z/OS, the *z/OS* tag should be specified in its declaration in the pipeline definition file.

4.2 Managing job dependencies

In a pipeline, jobs are declared to be part of stages, and stages are executed sequentially. To move to a next stage, all the jobs of the current stage needs to be completed. In some cases, it can drastically slow down the overall execution of a pipeline: for instance, a long-running task in the build stage may block a task that is performing a syntax-checking process that occurs in a subsequent stage. As there is no dependency between these tasks (the syntax-checking job doesn't need any data from the build job), the syntax-checking job could run earlier even if it defined in a later stage.

To solve this situation, a feature called *Directed Acyclic Graph*[11] was introduced in GitLab CI/CD pipelines. The purpose of this feature is to solve complex dependencies issues while speeding up the execution of pipelines, by optimizing the scheduling of jobs.

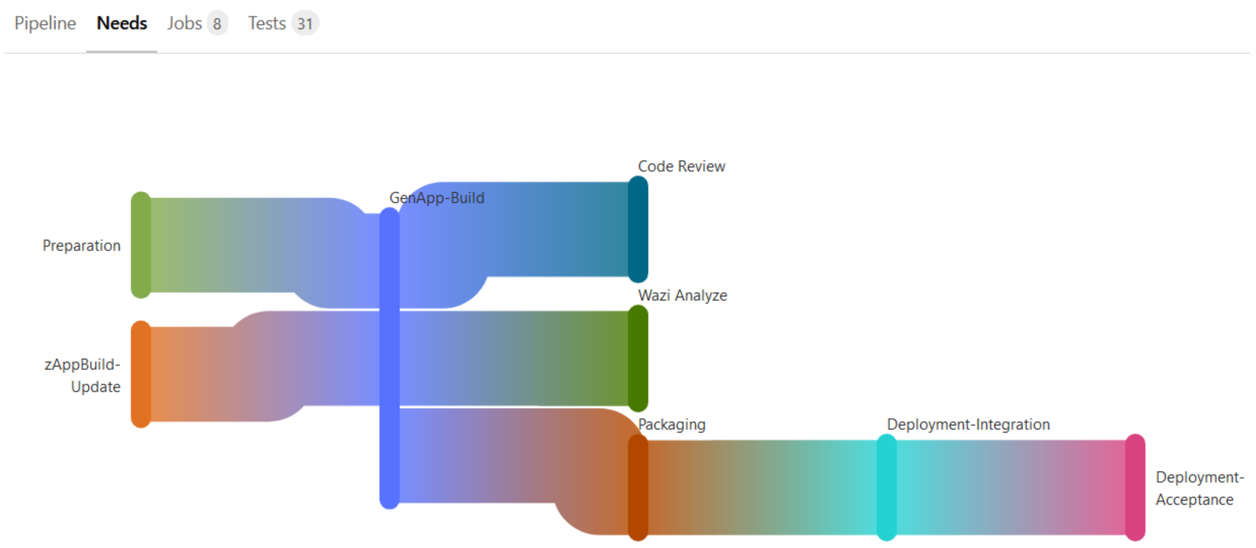
To benefit from this feature, a job must declare its execution dependencies using the *needs* keyword, providing a list of other jobs it depends on. For instance, consider the following pipeline definition:

```
stages:
    - Preparation
    - Build
    - Analysis
    - Packaging
    - Deployment-Integration
    - Deployment-Acceptance

zAppBuild-Update:
  stage: Preparation
  trigger:
    project: DAT/dbb-zappbuild
    branch: master
    strategy: depend

Preparation:
    stage: Preparation
    tags: [z/OS]
    script:
        - mkdir -p $CI_PROJECT_DIR/BUILD-$CI_PIPELINE_ID

GenApp-Build:
    stage: Build
    tags: [z/OS]
    needs: ["Preparation", "zAppBuild-Update"]

Code Review:
    stage: Analysis
    tags: [z/OS]
    needs: ["GenApp-Build"]

Wazi Analyze:
    stage: Analysis
    tags: [podman]
    needs: ["GenApp-Build"]

Packaging:
    stage: Packaging
    tags: [z/OS]
    needs: ["GenApp-Build"]

Deployment-Integration:
    stage: Deployment-Integration
    tags: [z/OS]
    needs: ["Packaging"]

Deployment-Acceptance:
    stage: Deployment-Acceptance
    tags: [z/OS]
    needs: ["Deployment-Integration"]
```

[11] https://docs.gitlab.com/ee/ci/directed_acyclic_graph/

In this configuration, the job *GenApp-Build* will require the jobs *Preparation* and *zAppBuild-Update* to be finished before executing. The next three jobs (*Code Review*, *Wazi Analyze* and *Packaging*) can be executed in parallel, however the *Code Review* job is using the same runner (tag is set to *z/OS*), so the *Packaging* job is in pending state, until the *z/OS*-tagged runner is released. The *Deployment-Integration* job needs the *Packaging* job, and the *Deployment-Acceptance* job needs the *Deployment-Integration* job.

A Directed Acyclic Graph is automatically built by GitLab for this configuration:



Depending on the availability of runners and dependencies relationships, the global execution of pipelines can be optimized when using this GitLab CI/CD capability.

4.3 Changing Git behavior

As part of every job in GitLab CI/CD, a specific step is managing the checkout of the project's repository to the target machine through the runner. However, in some situations, checking out the Git repository is not always wished, because you want to reuse the artifacts that were previously checked out or that were generated by a previous job. That's typically the case when leveraging build artifacts created by DBB, like the build report: this build report contains information about artifacts' locations and types for instance, that can be reused for packaging or for code review processing.

To change this default behavior of checking Git for every job, specific variables can be set either at the pipeline level or at the job level. The latter option is the preferred way for fine-tuning and manage the repository checkout granularly. The *GIT_STRATEGY*[12] variable controls how the GitLab CI/CD pipeline performs the Git checkout: when set to *clone*, the entire repository will be cloned for this job; when set to *fetch*, only the changes in the Git repository of the project are retrieved. When set to *none*, no Git operation is performed, and the project's workspace is left untouched.

[12] https://docs.gitlab.com/ee/ci/runners/README.html#git-strategy

The following pipeline definition shows how this configuration can be used:

```
Preparation:
    stage: Preparation
    tags: [z/OS]

GenApp-Build:
    stage: Build
    variables:
        GIT_STRATEGY: none

Code Review:
    stage: Analysis
    variables:
        GIT_STRATEGY: none

Packaging:
    stage: Packaging
    variables:
        GIT_STRATEGY: none
```

In this configuration, the project's repository is only checked out once, during the *Preparation* job. Subsequent jobs will reuse the artifacts that were retrieved at the beginning of the pipeline.

4.4 Adding manual jobs to the pipeline

Pipelines are usually made of automated processes but for some occasions, the execution must be stopped when a human action must be done, or a decision must be taken. This is typically the case in deployment pipelines, where moving an application to a QA stage or to production should be approved by a reviewer.

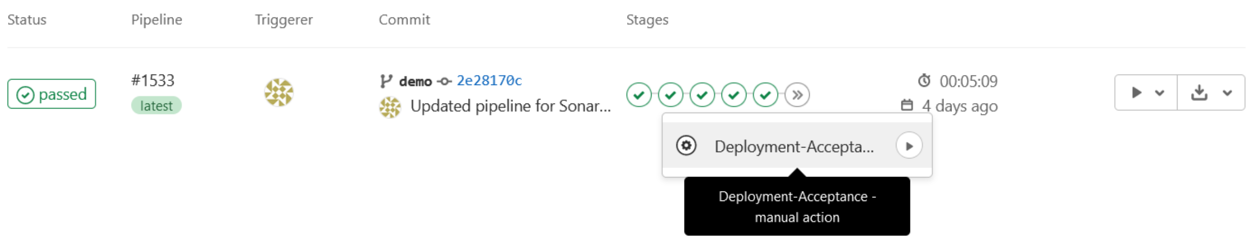
GitLab CI/CD provides the capability to design such pauses in the automated pipeline execution, by specifying manual[13] jobs. Manual jobs require attention before being scheduled, and the user actually decides when to start the execution of these jobs.

To specify a manual job, the keyword *when: manual* has to be specified in the job definition. In the following example, the *Deployment-Acceptance* job is set to be manual:

```
Deployment-Acceptance:
    stage: Deployment-Acceptance
    tags: [z/OS]
    needs: ["Deployment-Integration"]
    when: manual
    variables:
        GIT_STRATEGY: none
```

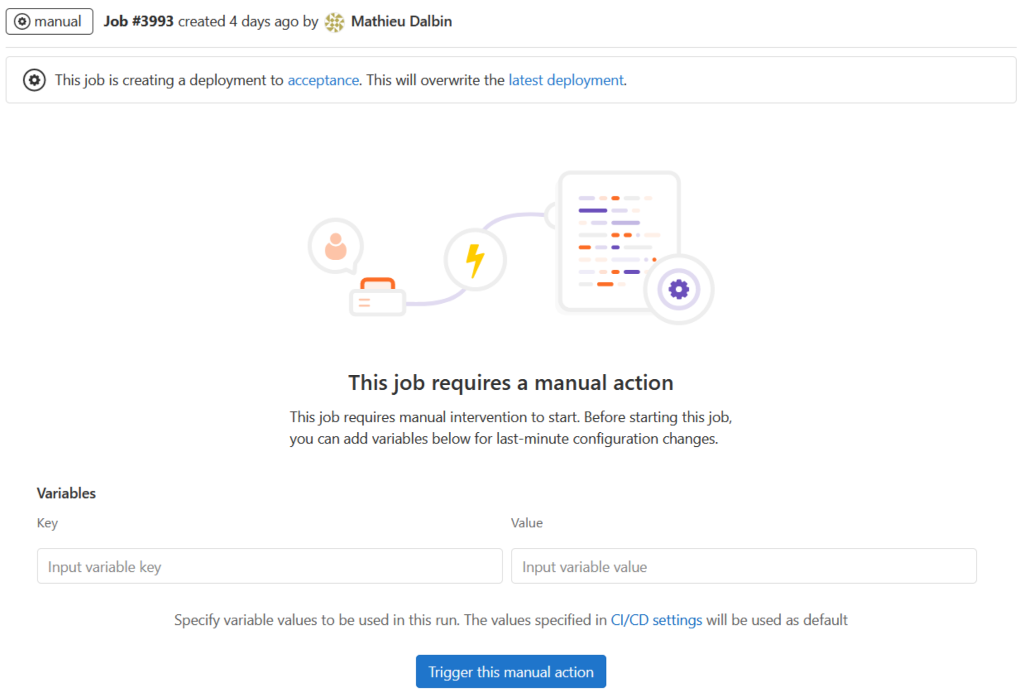
[13] https://docs.gitlab.com/ee/ci/yaml/#whenmanual

Even if all the conditions and requirements are met, this job will not be automatically started. The status of the job will be *manual*, and will require a specific action:



Status	Pipeline	Triggerer	Commit	Stages		
passed	#1533 latest		demo 2e28170c Updated pipeline for Sonar...		00:05:09 4 days ago	

The approver has the possibility to start all the manual jobs from the CI/CD pipelines view (with the *Play* button on the right side of each pipeline) or click on the manual job itself. In the latter option, the approver will be presented with this view, where additional variables can be specified if necessary:



manual **Job #3993** created 4 days ago by **Mathieu Dalbin**

This job is creating a deployment to acceptance. This will overwrite the latest deployment.

This job requires a manual action

This job requires manual intervention to start. Before starting this job, you can add variables below for last-minute configuration changes.

Variables

Key	Value
Input variable key	Input variable value

Specify variable values to be used in this run. The values specified in CI/CD settings will be used as default

Pressing the *Trigger this manual action* button will start the execution of this manual job.

4.5 Using GitLab Environments to automate cleanup processes

Environments in GitLab are a convenient way to track the deployment of applications. Each time a new version of the application is deployed, the corresponding environment in GitLab is updated to reflect that state. Environments can be dynamically created as part of the CI/CD pipeline, depending on the development needs.

Additional automation can be bound to environments, specifically when they are stopped. Several conditions can trigger the stop of environments: the deletion of a branch, the triggering of another job of the pipeline, or a user-defined timeout. These different configurations are described in GitLab's documentation[14]. In this sample configuration, it's the deletion of a branch which triggers the stop of the associated environment.

[14] https://docs.gitlab.com/ee/ci/environments/#stop-an-environment

A job can be executed when the environment stop request is emitted: this job can be used to perform specific tasks for housekeeping purposes. In this context, two specific groovy scripts can be leveraged:

- The DeletePDS[15] utility is used to delete a set of partitioned datasets (PDS) whose names match the pattern passed as parameter.
- The WebAppCleanUp[16] utility can delete the collections and the build result groups, passed as parameters when it is invoked.

These two utilities can be integrated in the pipeline to remove artifacts from z/OS, as well as the collections and the build results in the DBB WebApp, when they are not needed anymore. These scripts help to keep the build environment clean when all related activities are completed.

Leveraging these tools are particularly meaningful for short-living branches that are created to support specific implementations: feature branches or hotfix branches can benefit from these utilities for optimizing their footprint on z/OS and the DBB WebApp and minimizing storage usage.

In this sample configuration, the pipeline logic is extended with two jobs, which respectively create the environment (*Branch-Deployment* job) and perform cleanup tasks (*Branch-Cleanup-Host* job). While these two jobs depend on each other, it is necessary to define the same condition to execute them: in the below samples, the *only* keyword specifies to run these jobs only when branches are not protected and when there is no tag associated.

The definition for the *Branch-Deployment* job is as follows:

```
Branch-Deployment:
    stage: Branch Deployment
    tags: [z/OS]
    needs: ["Code Review"]
    only:
        variables:
            - $CI_COMMIT_REF_PROTECTED != "true" && $CI_COMMIT_TAG == null
    variables:
        GIT_STRATEGY: none
        GIT_CHECKOUT: "false"
    dependencies: []
    environment:
        name: branches/$CI_COMMIT_REF_NAME
        on_stop: Branch-Cleanup-Host
    script:
        - echo "Deploying environment branches/$CI_COMMIT_REF_NAME"
```

The *environment* keyword is used to specify the GitLab environment associated to our branch: in this case, the environment name will be the name of the branch prefixed by *branches/*. If the environment doesn't exist yet in GitLab, it is automatically created. The *on_stop* keyword defines the name of the job to be executed when the environment is stopped. In this sample configuration, the job *Branch-Cleanup-Host* is executed as part of the pipeline.

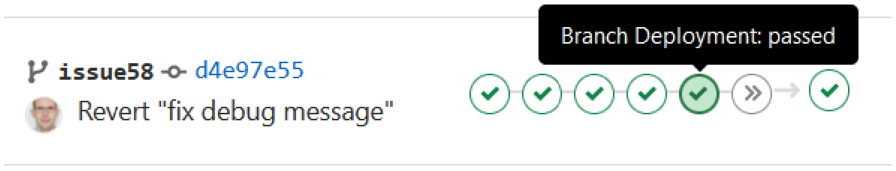
[15] https://github.com/IBM/dbb/tree/main/Utilities/DeletePDS
[16] https://github.com/IBM/dbb/tree/main/Utilities/WebAppCleanUp

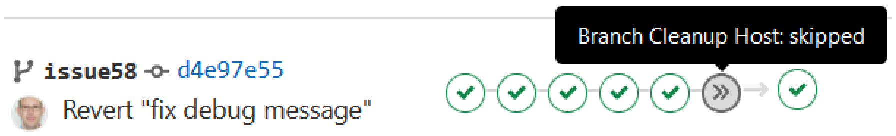
The *Branch-Cleanup-Host* job is defined as:

```
Branch-Cleanup-Host:
    stage: Branch Cleanup Host
    tags: [z/OS]
    needs: ["Branch-Deployment"]
    only:
        variables:
            - $CI_COMMIT_REF_PROTECTED != "true" && $CI_COMMIT_TAG == null
    variables:
        GIT_STRATEGY: none
        GIT_CHECKOUT: "false"
    dependencies: []
    environment:
        name: branches/$CI_COMMIT_REF_NAME
        action: stop
    when: manual
    script:
        - export LLQ=`echo ${CI_COMMIT_REF_NAME} | sed 's/-//g' | sed 's/^[0-9]*//g' | cut -c
1-8 | tr '[:lower:]' '[:upper:]'`
        - echo "Deleting datasets with HLQ ${HLQ}.${LLQ}"
        - $DBB_HOME/bin/groovyz $DBB_EXTENSIONS/cleanup/DeletePDS.groovy -h ${HLQ}.${LLQ}
        - echo "Cleaning up collections and build groups in DBB Web App"
        - $DBB_HOME/bin/groovyz $DBB_EXTENSIONS/cleanup/WebAppCleanUp.groovy --groups
$CI_PROJECT_NAME-$CI_COMMIT_REF_NAME --collections $CI_PROJECT_NAME-
$CI_COMMIT_REF_NAME,$CI_PROJECT_NAME-$CI_COMMIT_REF_NAME-outputs --prop
$DBB_EXTENSIONS/cleanup/DBB-GitLab.properties
```

The *environment* definition is similar to the previous job, except for the *action* keyword which has the *stop* value. When being triggered, this *Branch-Cleanup-Host* job will execute the scripted commands, which are leveraging the *DeletePDS.groovy* and the *WebAppCleanUp.groovy* scripts.

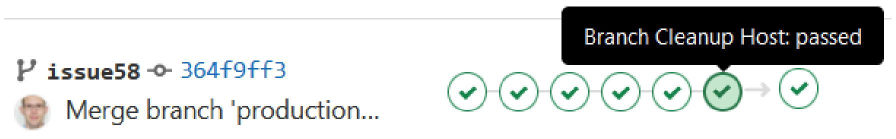
Based on the above configuration, the following workflow is implemented: When the pipeline is executed for a short-living branch, an environment is created at the first execution of this pipeline. This environment in GitLab will live as long as the branch exists. Here, the first execution of the pipeline for the branch *issue58* shows the following status:





While the *Branch-Deployment* job is executed and passed successfully (leading to the creation of the environment *branches/issue58* in GitLab), the *Branch-Cleanup-Host* job is skipped, which means it was not yet executed.

When the branch *issue58* is deleted after the changes got merged, the *Branch-Cleanup-Host* job is started and changes its status:



The execution log for this step shows the clean-up tasks that were performed on z/OS and on the DBB WebApp:

```
$ echo "Deleting datasets with HLQ ${HLQ}.${LLQ}"
Deleting datasets with HLQ GITLAB.CATMAN.ISSUE58
$ $DBB_HOME/bin/groovyz $DBB_EXTENSIONS/cleanup/deletePDS.groovy -h ${HLQ}.${LLQ}
** Deleting all datasets filtered with HLQ 'GITLAB.CATMAN.ISSUE58'
*** Deleting 'GITLAB.CATMAN.ISSUE58.BZU.BZUCFG'
*** Deleting 'GITLAB.CATMAN.ISSUE58.BZU.BZUPLAY'
*** Deleting 'GITLAB.CATMAN.ISSUE58.BZU.BZURPT'
*** Deleting 'GITLAB.CATMAN.ISSUE58.COBOL'
*** Deleting 'GITLAB.CATMAN.ISSUE58.COPY'
*** Deleting 'GITLAB.CATMAN.ISSUE58.DBRM'
*** Deleting 'GITLAB.CATMAN.ISSUE58.LOAD'
*** Deleting 'GITLAB.CATMAN.ISSUE58.OBJ'
*** Deleting 'GITLAB.CATMAN.ISSUE58.TEST.COBOL'
*** Deleting 'GITLAB.CATMAN.ISSUE58.TEST.LOAD'
** Deleted 10 entries.
** Build finished
$ echo "Cleaning up collections and build groups in DBB Web App"
Cleaning up collections and build groups in DBB Web App
$ $DBB_HOME/bin/groovyz $DBB_EXTENSIONS/cleanup/WebAppCleanUp.groovy --groups
$CI_PROJECT_NAME-$CI_COMMIT_REF_NAME --collections $CI_PROJECT_NAME-
$CI_COMMIT_REF_NAME,$CI_PROJECT_NAME-$CI_COMMIT_REF_NAME-outputs --prop
$DBB_EXTENSIONS/cleanup/DBB-GitLab.properties
Creating repository client for https://dbb.dat.ibm.com:11443/dbb
** Deleting build groups: [catalog-manager-issue58]
*** Deleting build group 'catalog-manager-issue58' -> Status = HTTP/1.1 200 OK
** Deleting collections: [catalog-manager-issue58, catalog-manager-issue58-outputs]
*** Deleting collection 'catalog-manager-issue58' -> Status = HTTP/1.1 200 OK
*** Deleting collection 'catalog-manager-issue58-outputs' -> Status = HTTP/1.1 200 OK
** Build finished
```

5 Extending the pipeline with new capabilities

The following samples are provided to highlight how to extend the pipeline with z/OS-related capabilities. The provided extracts will require tailoring prior being integrated in your environment and the build process for Mainframe application.

5.1 Performing a syntax-check analysis with Code Review

5.1.1 Introduction to Code Review

Code Review, also known as Software Analysis, is a feature brought by IBM Developer for z/OS (IDz) and helps to inspect the quality of source code for Cobol and PL/I. Code Review is accessible from IDz, but can also be invoked in a pipeline. This latter option is leveraging a z/OS application that can be called in a batch process: this application is often referred as a *"headless Eclipse"*, as it has no UI. More information is available in the IBM Documentation[17].

The Code Review function works with a set of rules that can be enabled on-demand, and user-written rules can also be integrated in the analysis. Typically, this set of rules is edited through IDz and exported either through a Rule file (for standard rules) or a CCR file (for custom rules). These files are to be used in the Code Review process occurring on z/OS, as part of the pipeline.

To run the z/OS Code Review application, the preferred option is to submit a JCL that is calling a defined PROCEDURE, shipped with IDz. This PROCEDURE, called AKGCR, is provided through the IDz sample procedure library on z/OS, and is running a REXX script that starts the Code Review process. To automate the dynamic creation of the Code Review JCL and its submission to z/OS, the best option is to set a groovy script in place. Additionally, a sample script is available in the public GitHub repository for DBB[18] to serve that purpose.

5.1.2 Integrating Code Review into the GitLab CI/CD pipeline

To integrate the Code Review process in the pipeline execution, prior customization must be performed. The *RunCodeReview* groovy script works with a property file, called *codereview.property*, in which some tailoring is required. For instance, a valid JOBCARD will need to be specified and other Code Review-related properties can be filled in. Some properties can be specified as input parameters when invoking the *RunCodeReview.groovy* script.

The next phase is to add a job definition in the pipeline. The Code Review script will actually search for a DBB Build Report to parse, in order to extract the list of artifacts on which to apply the Code Review process. The *RunCodeReview* script will need to know the path of the work directory, where the Build Report is stored (on USS). This work directory is typically the same folder used by DBB, where all the build output artifacts are stored: this path is passed to zAppBuild through the *--workDir* parameter. When invoking the Code Review process, the same path will be passed, to ensure the Build Report is found.

[17] https://www.ibm.com/docs/en/developer-for-zos/15.0.0?topic=zos-code-review-environment

[18] https://github.com/IBM/dbb/tree/master/Pipeline/RunIDZCodeReview

The following lines are a sample job definition in the GitLab CI/CD pipeline file:

```
variables:
    DBB_HOME: "/usr/lpp/dbb/v1r0"
    DBB_EXTENSIONS: "/var/dbb/extensions"
Code Review:
    stage: Analysis
    tags: [z/OS]
    variables:
      GIT_STRATEGY: none
    script:
      - cd $CI_PROJECT_DIR/BUILD-$CI_PIPELINE_ID/build*
      - export BUILDPATH=`pwd`
      - $DBB_HOME/bin/groovyz $DBB_EXTENSIONS/idz-codereview/RunCodeReview.groovy --workDir
$BUILDPATH --properties $DBB_EXTENSIONS/idz-codereview/codereview.genapp.properties
```

With this definition, the Code Review process will occur in an *Analysis* stage and will execute on the z/OS environment. The Code Revie application is able to generate reports in different formats, including jUnit compatible reports that can be leveraged with additional reporting solutions.

5.2 Adding application insights with Wazi Analyze

When an application change is requested, providing valuable insights to the developer is key to ensure the good quality of the delivered features. Wazi Analyze helps providing the necessary information to perform impact analysis and have a better understanding of the relationships between applications components. Integrating this analysis in the pipeline can ease the life of developers, who will automatically have a refreshed vision of the application’s structure as they perform changes.

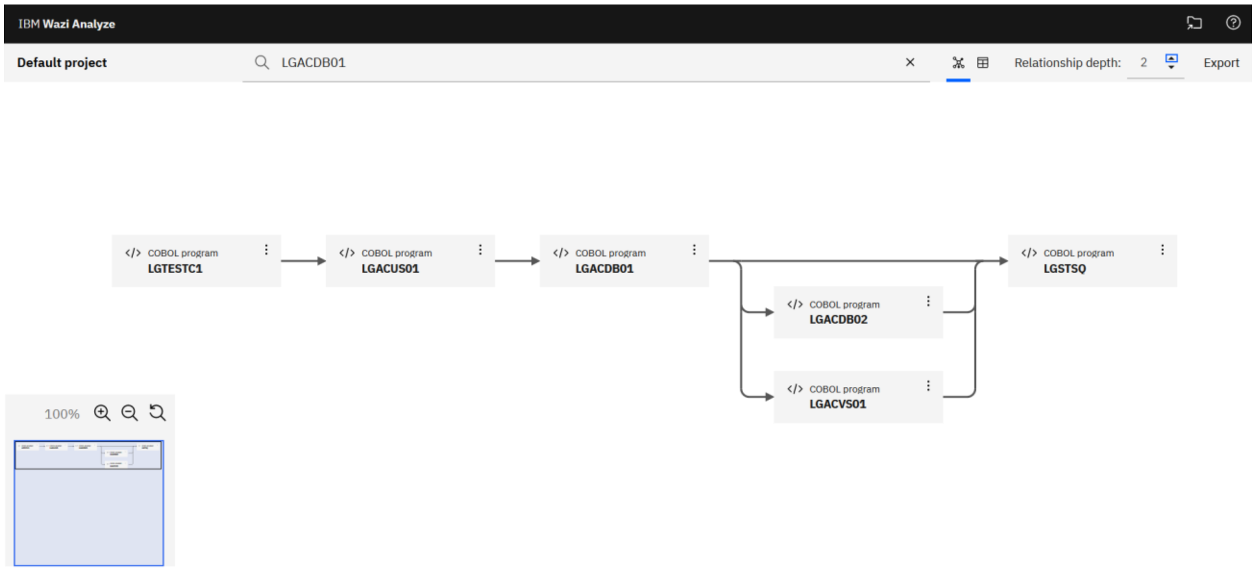
Wazi Analyze is shipped as a container (at the time of writing this document) which can be deployed using docker or a compatible solution. As part of the pipeline, this container cannot execute on the developer’s machine, but should be started on a close location related to the pipeline execution: in our use case, a new GitLab runner will be used to start the container, which will run on a Linux machine (x86).

Some actions in the Wazi Analyze container must be automated, to start the scan and the startup of the necessary services. The pipeline job will actually copy the application’s sources into a specific location that will be mounted into the container and will also generate a shell script to automate the necessary tasks. The *Analysis* stage used in Code Review will be leveraged to position this step in the overall pipeline.

The following sample job definition is performing all the necessary steps to copy the sources to the right location, start the container and run the analysis:

```
Wazi Analyze:
    stage: Analysis
    tags: [podman]
    script:
        - if [ "`sudo podman container ls -a | grep GenApp | wc -l | awk '{printf $1}'`" -gt 0
]; then WaziContainer=`sudo podman container ls -a | grep GenApp | awk '{printf $1}'`; sudo
podman container rm -f $WaziContainer; fi;
        - mkdir -p $CI_PROJECT_DIR/BUILD-$CI_PIPELINE_ID/source;
        - for f in `find $CI_PROJECT_DIR/genapp -iname "*.cbl"`; do mv $f
$CI_PROJECT_DIR/BUILD-$CI_PIPELINE_ID/source/"$(basename ${f%} | tr [:lower:] [:upper:])";
done
        - for f in `find $CI_PROJECT_DIR/genapp -iname "*.cpy"`; do mv $f
$CI_PROJECT_DIR/BUILD-$CI_PIPELINE_ID/source/"$(basename ${f%} | tr [:lower:] [:upper:])";
done
        - for f in `find $CI_PROJECT_DIR/genapp -iname "*.bms"`; do mv $f
$CI_PROJECT_DIR/BUILD-$CI_PIPELINE_ID/source/"$(basename ${f%} | tr [:lower:] [:upper:])";
done
        - for f in `find $CI_PROJECT_DIR/genapp -iname "*.jcl"`; do mv $f
$CI_PROJECT_DIR/BUILD-$CI_PIPELINE_ID/source/"$(basename ${f%} | tr [:lower:] [:upper:])";
done
        - touch $CI_PROJECT_DIR/BUILD-$CI_PIPELINE_ID/source/analyze.sh
        - echo "#!/bin/sh" >> $CI_PROJECT_DIR/BUILD-$CI_PIPELINE_ID/source/analyze.sh
        - echo "" >> $CI_PROJECT_DIR/BUILD-$CI_PIPELINE_ID/source/analyze.sh
        - echo "/home/wazianalyze/script/wa-scan.sh" >> $CI_PROJECT_DIR/BUILD-
$CI_PIPELINE_ID/source/analyze.sh
        - echo "sleep 5" >> $CI_PROJECT_DIR/BUILD-$CI_PIPELINE_ID/source/analyze.sh
        - echo "/home/wazianalyze/script/wa-startup.sh" >> $CI_PROJECT_DIR/BUILD-
$CI_PIPELINE_ID/source/analyze.sh
        - chmod 755 $CI_PROJECT_DIR/BUILD-$CI_PIPELINE_ID/source/analyze.sh
        - sudo podman run -d -v $CI_PROJECT_DIR/BUILD-
$CI_PIPELINE_ID/source:/home/wazianalyze/data/project/source -it -p 5000:5000 --name GenApp
localhost/ibmcom/wazianalyze:1.2.0 sh -c "/home/wazianalyze/data/project/source/analyze.sh &&
while true; do sleep 5; done"
        - echo "Wazi Analyze available at https://gitlab.dat.ibm.com:5000/explore"
```

At the end of the job execution, the developer can open the Wazi Analyze UI, with the link provided in the output log of the job. Through this web-based solution, analysis can be performed with the latest versions of the artifacts that were used during the build, ensuring consistency between the structure of the application and the actual load modules that were built.


IBM Wazi Analyze
Default project
LGACDB01
Relationship depth: 2
Export
COBOL program LGTESTC1
COBOL program LGACUS01
COBOL program LGACDB01
COBOL program LGSTSQ
COBOL program LGACDB02
COBOL program LGACVS01
100%