CI guide for BE-CEM-EDL
(for Linux OS)

Chapter 1 - Introduction

- ***What is Continuous Integration***

By definition, CI is a software engineering practice where developers can automate the merge of code changes from different branches, into a single project. Automated builds and tests are run to ensure that there is no issue, or for better bug traceability. A version control system is at the core of the CI process. The version control system is also supplemented with other checks like code quality tests, syntax style review tools, and more. The process contains several automation tools that emphasize code correctness.

Some of the main practices of continuous integration are the following:

1. Maintain a single project repository
2. Automate the build phase to make it self-testing
3. Keep it as fast as possible
4. Every commit should be built on an integration machine
5. Make it easy for any user to get the latest executable version of the project version

By following these rules, elimination of the chance for catastrophic merge issues can be achieved.

- ***How does this work***

Developers, usually check out code, into their private workspace. When done, they commit the changes to the repository. With a CI service like GitLab, servers monitor the repository and automatically compile, build, and test every new version of code committed and ensure that the entire team is alerted any time the central code repository contains broken code. However, CI doesn’t get rid of errors, but it does make them productively easier to find and remove.

In BE-CEM-EDL, both GitLab (gitlab.cern.ch) and OHWR (ohwr.org), Open Hardware Repository, are used for source code version management. The latter is a GitLab community edition which is being used by BE-CEM-EDL mostly alongside with GitLab enterprise edition. Therefore, the same tools and mechanism can be applied for repositories in ohwr.org and in GitLab. For convention reasons, in this documentation, GitLab refers to gitlab.cern.ch and OHWR refers to ohwr.org. One of the most preferred ways to build and run tests for each project is to use Docker alongside with GitLab CI.

- ***Docker***

Since Docker is an important software tool which is used for the creation and improvement of the CI process, here is some significant information about it. Docker is an open source platform which permits developers to package applications into containers. They are standardized executable components combining application source code with the operating system (OS) libraries and dependencies required to run that code in any environment [1]. Containers are executable units of software in which application code is packaged, along with its libraries and dependencies, in common ways so that it can be run anywhere, whether it be on desktop, traditional IT, or the cloud [2].

Some of the major tools and terminology that are often used and encountered when using Docker are:

1. Dockerfile : Text file consisting of instruction commands needed to build the docker container image. In this way the creation process of this image can be automated
2. Docker Image : Contains executable application source code as well as all the tools, libraries, and dependencies that the application code needs to run as a container. When you run the Docker image, it becomes one instance (or multiple instances) of the container [1]. A docker image can be created from scratch or can be pulled down from common repositories.
3. Docker Containers : Docker containers are the live, running instances of Docker images. They are live, temporary, executable content. Users can interact with them, and adjust their settings and conditions using docker commands.

- ***GitLab CI***

As already mentioned, GitLab (and OHWR), provides a full CI/CD environment for the users who want to trace bugs and errors. The majority of the concepts can be described below:

1. GitLab Runner : an application that works with GitLab CI/CD to run jobs in a pipeline
2. Pipelines : the top-level component of CI. They comprise the jobs and stages which define what to do and when, respectively
3. Job Artifacts : the output of jobs that can either be an archive of files or directories

In order to get started with GitLab CI, there are some prerequisites that need to be met. First of all, the user must have the maintainer or owner role of the project. In addition, the latest edition of GitLab-runner should be installed. In this way, one or more runners can be registered in the project, since there is no limitation in the number of runners. Once a job is triggered, an activated registered runner is going to run the job. In conclusion, the creation of a *.gitlab-ci.yml* file is needed at the root of the repository. This is the file where all the CI jobs are defined.

In the Figure 1, there is an architecture diagram which describes briefly the idea of continuous integration with Gitlab. From one side there is the Gitlab Instance, which as already mentioned, can be either gitlab.cern.ch or ohwr.org and from the other side there is the server which has the gitlab-runners and runs the CI jobs. This server can be either a private server, like a real machine which can be handled remotely or a virtual machine (personal or shared project in openstack.cern.ch). It is very important for each project to have one file that describes the CI pipeline among with other

configuration of them. Then, different gitlab-runners will run different jobs, like simulations, synthesis etc.

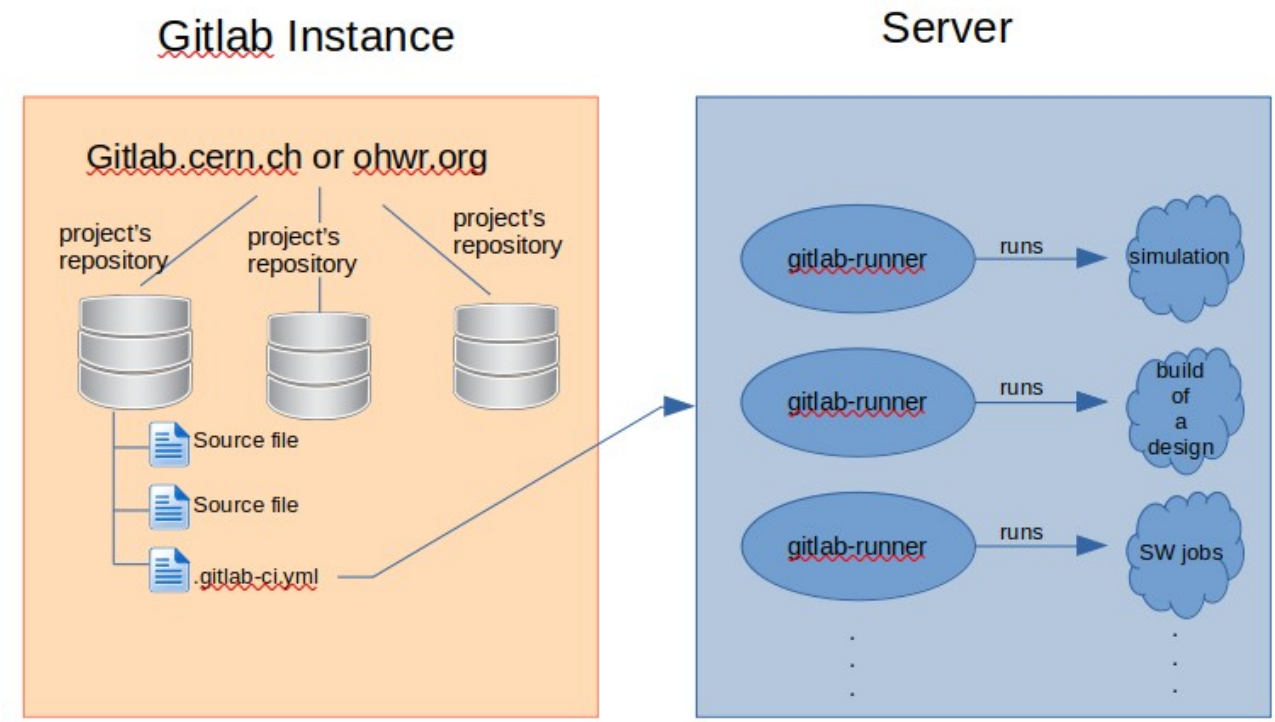

Figure 1. Architecture diagram of Gitlab CI

Subsequently, in the Figure 2, there is a flow diagram that shows, in briefly, the steps that a user need to follow, in order to use CI service in Gitlab. In the next chapters of this guide, all these steps are described with more details, in order to be clear for the user to better understand each step. Some important things to be declared at that point are:

- Docker images can be generated by the user or the user can pull some already existing, for example for a container registry like the one here in gitlab.cern.ch https://gitlab.cern.ch/cce/docker_build/container_registry
- The steps are the same either the project is at OHWR.org or at Gitlab.cern.ch. The only difference is that they can not use the same gitlab-runners.

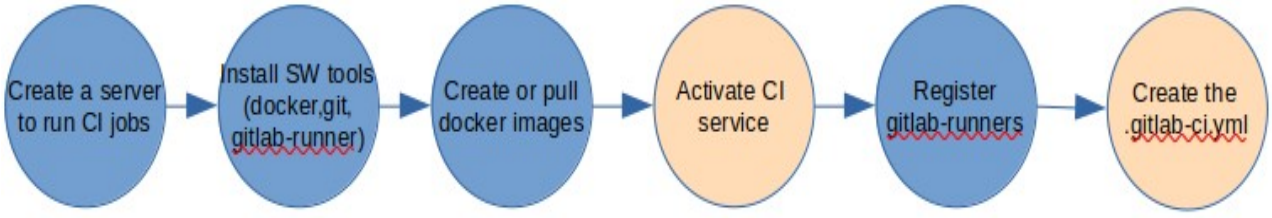

Figure 2. Flow diagram of the steps that the user need to follow

With the blue color in the background, there are the actions that the user should do in the server side and with the lighter color there are the steps in the Gitlab instance side.

Chapter 2 – Set up CI in a private server

In this chapter, the steps to set the whole CI environment in the Gitlab instance are presented. Through these steps, the user will be able to create docker images, Gitlab runners and start using continuous integration in every project. At this point, it is worth mentioning that the user that registers the runners, has to be a *maintainer* in the repository, in order to enable them each time.

- ***Step 1) Download prerequisite software tools***

The maintainer of the project, who will activate up the CI service and create the remote server, should download some software tools in the server which are needed. Below there is a list of the most important in order for someone to get started:

1. docker
2. git
3. gitlab-runner

Note: More details on how to install these tools can be found in Chapter 3. These steps can vary regarding the linux distribution that the server has.

- ***Step 2) Create the dockerfile***

The steps 2 to 4, can be skipped if the user will choose to use a docker image provided by some common repository. In case a new one will be created, a Dockerfile must be described, which is responsible for the generation of the docker image. One good practice, is to have a separate Dockerfile for different applications. In addition, all the scripts or other files should be present in the same directory where the dockerfile is.

In order to have a better view on how to write dockerfiles, you can always refer to the official docker documentation page [3]. However, below there is an example on the form that it must follow (Figure 3). The example is a dockerfile for Xilinx ISE 14.7. The full content can be found in the GitLab repository where all the custom docker images are stored [4].

```
FROM  ubuntu:18.04                                # specify the OS
SHELL ["/bin/bash", "-c"]                         # specify the shell environment

WORKDIR /root/                                    # the working directory
RUN dpkg --add-architecture i386 && apt-get update  # add the architecture (i386)

RUN apt-get install -y <tool/library/package>       # use RUN command to execute commands

COPY Xilinx /opt/Xilinx                           # copy existing files/directories into the new generated docker image
COPY setup_ise147.sh /root/                       # copy executable scripts from current directory
RUN source /opt/Xilinx/14.7/ISE_DS/settings64.sh    # execute scripts

COPY ./entrypoint.sh /                            # copy the entrypoint.sh script to docker image
ENTRYPOINT ["/entrypoint.sh"]                     # this is the end of the dockerfile where we specify which is the entrypoint
```

Figure 3. Dockerfile example for Xilinx ISE docker image

Briefly, a specific flow must be followed where in the beginning there is the declaration of the OS and other fundamental information about the image. Then, the required files in order to install the application, are copied into the Docker Image. In the end, the entrypoint script is also copied and specified. This entrypoint, is the script that will run first and can usually be a script that checks if the user of CI has all the rights to use it.

Note: In some cases, the programs that are used, can be quite big in terms of disc space. For that reason, it would be useful to use techniques to decrease the space that they require. One simple technique can be to just delete the files needed to install the program, after its installation.

- ***Step 3) Essential scripts needed***

The development of some scripts used from the Dockerfile is needed before the generation of the docker image :

1. *build_dockerfile.sh* : A script containing the command to build the docker image (docker build -t <image_name:version> .)
2. *entrypoint.sh* : Can be used for all projects. Contains a test where we check if the user is authorized to use CI.
3. *setup.sh* : A script containing all the appropriate commands in order to execute the application program, like Licenses and libraries path.

If needed, more scripts can be developed, so as to automate some steps and speed up the whole process. They have to be in the same directory as the Dockerfile too.

- ***Step 4) Generate the docker image***

In the same directory where Dockerfile and all the scripts and the program installation files are, we can execute the *build_dockerfile.sh.* It may take some time, depending on the server and the size of the docker image.

Every command from the Dockerfile, is being executed serially as it can be seen in the terminal. In case of an error at this stage, the docker image that is being created in the meanwhile, should be deleted (*docker rmi -f <image_name>*). Once the issues have been resolved, the image has been created. To view all the current images in the server, run *docker images.*

- ***Step 5) Activate the CI service***

The maintainer or the owner of the project, should activate in the main page of the repository, the CI service (Set up CI/CD). This can be done by checking the :

settings > general > permissions > check pipelines

Note: If the project exists, in addition, it may needed to check the .gitmodules files in the repository, if the submodule URL of the dependencies are clear.

- ***Step 6) Register Gitlab-runner(s)***

In order to create the gitlab-runner (after its installation -IMPORTANT to install it from the official repository of gitlab and not from Ubuntu's apt), the first step is to register a runner. This can be done from the server machine. This is where all the docker images, gitlab-runners and everything needed for CI/CD is installed. The command for register a new gitlab-runner is:

- *(sudo) gitlab-runner register*

For the registration, some information must be provided by the user. These are, the instance, for example gitlab.cern.ch or ohwr.org, the registration token, as it can be seen in the settings → CI → runners, as it can be seen in the Figures 4 and 5.

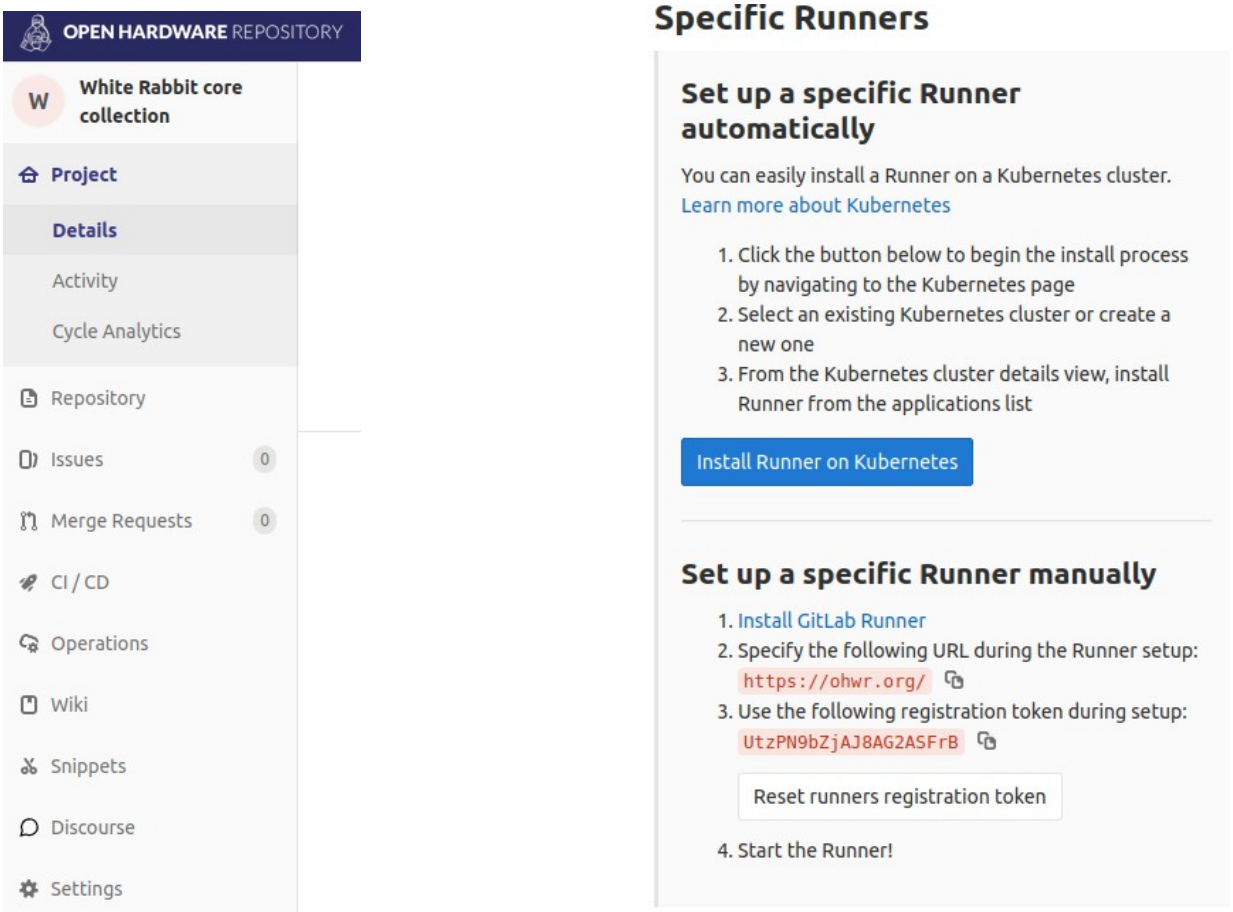

Figure 4 and 5. Settings tab in ohwr.org project and where to find the URL and token needed

Subsequently, define the description and the tag of the runner and select the executor (docker is the mainly used one) and the default docker image (the one created in step 4). In the end, when the runner has been registered, the final step is to enable and verify the runner with these commands:

- `(sudo) gitlab-runner start`
- `(sudo) gitlab-runner verify`

In settings→CI→runners, we need to check if the runner (with the specific tag) is not paused. By that time, CI is ready to start.

- ***Step 7) config.toml tips***

This is the configuration file where the runners can be modified once they are registered. The path for this is: */etc/gitlab-runner/config.toml*, or in some cases it can be found in: *~/.gitlab-runner/config.toml*

It can happen that, the log file probably exceeds the default maximum limit, which is 4MB. To change it, this field must be modified manually, as it is shown in Figure 6:

```
[[runners]]
  output_limit = 20000
  [runners.docker]
    volumes = ["/cache"]
    pull_policy = ["always", "if-not-present"]
```

Figure 6. Useful additions to the config.toml file

In main web page of GitLab, more information regarding all the config.toml file can be found [6].

Note: After every alteration in the runner, always run *gitlab-runner restart* and *gitlab-runner verify*.

- ***Step 8) .gitlab-ci.yml file development***

Another important step is the development of the .gitlab-ci.yml file in the root directory of the project's repository. This file will have all the pipelines, jobs and in general everything needed to be executed. For HDL gateware projects, two are the most important stages. These are the simulation and the whole build of the design. Specifically, in this file you can define [5]:

- The scripts you want to run.
- Other configuration files and templates you want to include.
- Dependencies and caches.
- The commands you want to run in sequence and those you want to run in parallel.
- The location to deploy your application to.
- Whether you want to run the scripts automatically or trigger any of them manually.

The basic structure of this file can be seen below, in the Figure 7, with the very fundamental structure :

```
variables:
  GIT_SUBMODULE_STRATEGY: normal
  GIT_DEPTH: "1"

stages:
    - simulation
    - build
```

```
wb_fine_pulse_gen_simulate:
  tags:
  - modelsim_10_2a
  stage: simulation
  script:
    - /entrypoint.sh
    - source ~/setup_modelsim.sh
    - cd testbench/wishbone/wb_fine_pulse_gen
    - git submodule init && git submodule update
    - cp /opt/compiled_libs_ise14.7/modelsim.ini .
    - hdlmake makefile
    - cat Makefile
    - make
    - vsim -c -do run.do
  artifacts:
    paths:
    - hdl/testbench/wishbone/wb_fine_pulse_gen
```

Figure 7. Simple example of a simulation CI job

A complete guide of the available features can be found in the gitlab webpage [5]. In a few words, first there is the variable and the stages declaration. By grouping some of the jobs in stages, there is an opportunity to specify the sequence that they will run. In this example, the simulation jobs will start and if they pass with no errors, then the build stage jobs will follow.

Each job has a name to describe what they do, a tag to refer to a specific Gitlab-runner and the stage that they belong. In addition, there is of course the script part, where there are the commands that serially will be executed. Lastly, another important thing to declared is the artifacts and especially the path that exist the result(s) of each successful CI job.

- ***Step 9) Final step***

At this final step everything is set up and the GitLab CI is ready. Depending on the configuration that the owner/maintainer of the project prefers, every time that a push triggers the CI system, the jobs start running. The status of them can be seen in the tab on the left of the project's start screen, CI/CD → Pipelines.

Chapter 3 – Set up CI in a personal Openstack project

Openstack is an open source cloud computing software [7]. This cloud service can be used instead of a private server. In this way, the provided resources are used by the user to create a cloud server which will host the CI infrastructure. After the creation of this cloud server, the steps are the same as the private server ones.

- ***Step 1) Activate the openstack from CERN***

First of all, the user should subscribe to the cern cloud infrastructure. After that, go to the webpage https://openstack.cern.ch, where the user opens a ticket in order to request for a personal project. Unfortunately, this personal project has some default resources like 250Gb of disc space, and 20Gb of RAM which can be used to create up to 5 virtual machines (instances, as they are called). More specifically, every virtual machine can have the resources that the user defines.

This approach, can be useful in smaller and less demanding projects and can be a testing server for everyone who wants to experiment with CI and try different things. In addition, servers like them, can be used to run simulation jobs (for gateware), simple software jobs and also build of lightweight designs.

- ***Step 2) Create a linux virtual machine***

For this step and for all the following, there is also another documentation that the reader can refer to, in order to read in more depth the whole process and learn more things about Openstack.

The first thing after the subscription and having created the personal project in Openstack, is to create a (Linux) virtual machine, to host your server. For that reason, a keypair is needed, which can be generated from an lxplus/lxplus8 account with ssh `ssh-keygen -t rsa -f cloud.key`. Then, this keypair must be imported. In the main menu of openstack webpage the flow is from the tab on the left to go *Project* > *Compute* > *Key pairs* > *Import public key.*

There, a name should be given (e.x. lxplus) and the key type should be SSH KEY, before submitting the public key. Different virtual machines can use the same public key, so this public key can be used by others too.

Subsequently, in the same menu and in the tab *Project* > *Compute* > *Instances* > *Launch Instance* is where the user can create the Linux virtual machine and choose the resources that it will have. The latest stable Linux version is the CentOS 8 (C8-x86_64 [2021-12-01]) among the others that are provided in the list. When everything is set up, the machine is ready to be launched. If there are no errors in the process, the personal server can be accessed by: `ssh root@VM_NAME` or `ssh -i cloud.key root@VM_NAME`. In the Figure 8, the window is shown that opens when an instance is launched. Briefly, the user should provide the details of the instance, the source, the flavor (the machine's RAM), import the key and optionally, in Metadata tab, to select and activate CERN's metadata.

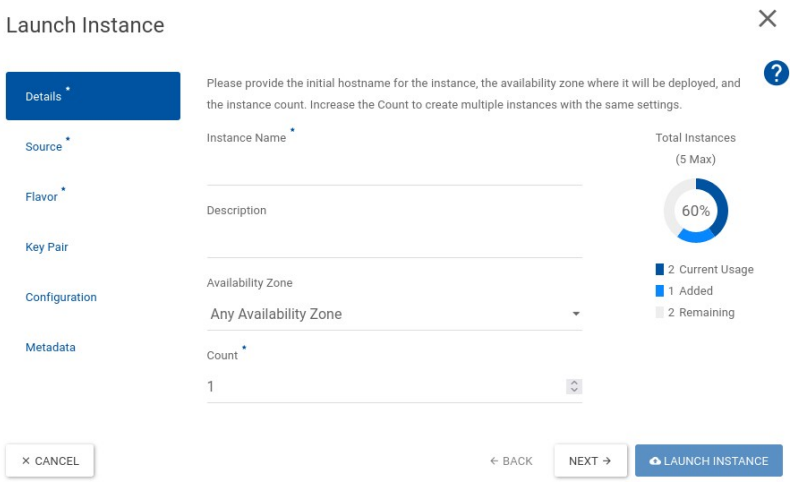

Figure 8. Launching an instance

- ***Step 3) Create a volume***

A volume is an arbitrary sized disk to be attached to your virtual machine, like plugging in a USB stick. It is necessary, because by default, the server has limited disc space resources. From the menu on the left of the web page *Project* > *Volumes* > *Create Volume* and again the name, description and size are the most important fields to be filled. Once it is created, it should be attached and then mounted to the specific virtual machine. As it is shown in the Figure 9 below, these are the appropriate fields:

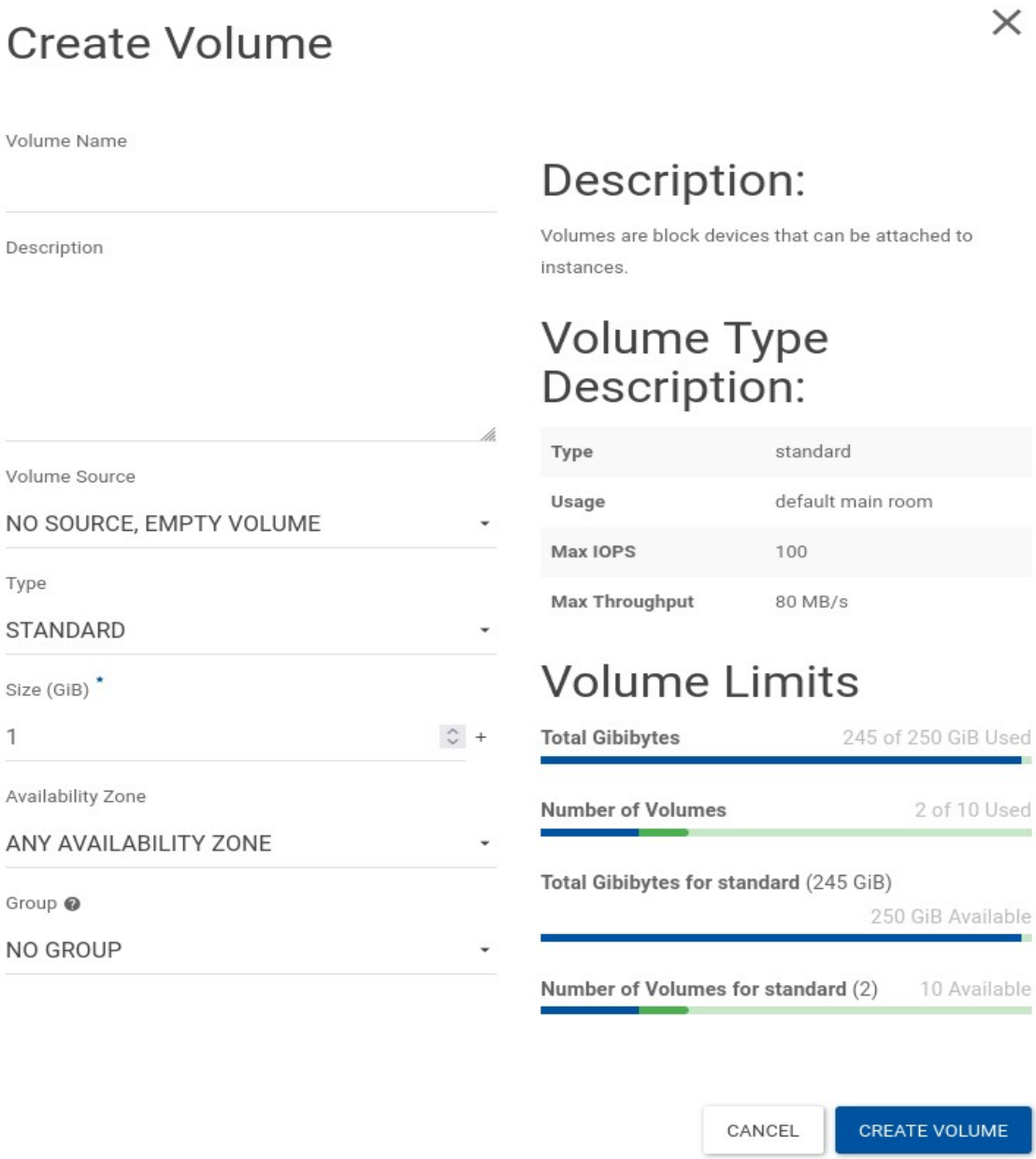

Figure 9. Creation of a volume

First of all, name and short description must be provided, the type (it can be standard, io1, etc) and the size.
Once the volume has been created, it should be mounted to a virtual machine. This can be done from lxplus and inside the virtual machine, with the help of the following commands:

`openstack volume list` (to check that it has been created)

`openstack server add volume my-vm my-volume` (to attach the volume to VM)

This action can be done also in the graphical environment and not only through terminal. Just by clicking “Manage Attachments” in the specific volume. Then, the rest of the process is being done inside after login to the virtual machine, as described in the previous step:

`cat /proc/partitions` (to verify that it is already attached)

`mkfs -t ext4 /dev/vdb`

`mount /dev/vdb /mnt` (to mount it)

`df -H` (check that is has mounted)

Note: Since this is a personal project in openstack.cern.ch, the resources are by default specified, so there is no option to request for more (something that can be happen in case of a shared project).

- ***Step 4) Set up the server***

At this stage, all the previous steps can be followed, like the private server case (Chapter 2). Briefly, what is needed is to install *docker*, *git* and *gitlab-runner*. In addition, the docker images should be stored and the runner(s) should be registered.

Below, there is a step by step guide on how to install everything needed in these CentOS virtual machines:

- Install git:
  - dnf update -y
  - dnf install git -y

- Install docker:
  - `yum install -y yum-utils`
  - `yum-config-manager \`
    `--add-repo https://download.docker.com/linux/centos/docker-ce.repo`
  - `yum install docker-ce docker-ce-cli containerd.io`

Note: It is suggested that the docker images should be stored in the attached volumes to the virtual machine, because they demand more disc space than the provided by default. In order to change the directory where the images will be stored, these are the actions needed:

- mkdir -p /etc/systemd/system/docker.service.d
- vim /etc/systemd/system/docker.service.d/docker-storage.conf
  And in this file these are the changes that need to be applied:

```
[Service]
ExecStart=
ExecStart=/usr/bin/dockerd -H fd:// --data-root="/mnt"
```

- systemctl daemon-reload
- systemctl restart docker
- docker info|grep "Docker Root Dir" (it should point to the new directory)
- rm -rf /var/lib/docker (safely remove old Docker storage)

- Install gitlab-runner
  - `curl-L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.rpm.sh | sudo bash`
  - `yum install gitlab-runner`

Note: If the user wants to use some docker images that are stored in another private server and transfer them in this personal openstack project, there are some hints in order to achieve that. In the virtual machine, the key of the other server should be added in the .ssh/authorized_keys. Then, the docker image can be easily transferred by using this command:

```
docker save IMAGE_ID | gzip | ssh root@VM_NAME ‘gunzip | docker load ‘
```

At this point, it is important to be mentioned, that docker images for various EDA tools can be found in this Gitlab repository https://gitlab.cern.ch/cce/docker_build/ made by EPC section. Then, in the “Packages & Registries” and in “Container Registry”, there is a list of the most common used docker images. The steps to pull these images are explained in the README file of the repository.

Chapter 4. Set up in a shared openstack project

Here, the process is quite similar with the one in the previous chapter. One main difference is that in this shared project doesn’t have a limitation in terms of the available resources. This is a useful asset because in one project, there can be multiple virtual machines and in this way, it can be maintained easier. The user can request for more resources from IT, just by clicking on “*Request a quota change*”, in the main web page of the shared project in Openstack. In addition, there is the chance, anytime, to change the owner of this shared project, since it can be inherited from one user to another, or can have multiple owners.

In order to create this shared project in Openstack, the user should open a ticket, like in the case of the personal project, but must predefined that this is a shared project and also request the initial resources. Another important thing that must be included in the ticket is to mention the maximum value of the RAM. By default it is *large* which is 7.5Gb, so it can be up to 15Gb (*xlarge*). This can change in the future or even in the initial stage of the creation but it is a little more complicated in terms that a more detailed report needs to be provided to IT, explaining the reason that something like this is useful.

The steps from 1 to 4, as they described in the previous Chapter are the same here and can be applied in order to create virtual machines. Once everything has set up and the infrastructure is

ready, the user doesn't need to change or maintain anything in this cloud server. The following step is to go to the project's repository (either it is in ohwr.org or gitlab.cern.ch) and activate the CI/CD option, alongside with some configuration needed from *Settings > CI/CD > General Pipelines*, like Timeout and Custom CI config path to be always .gtilab-ci.yml file. The final stage is to create the gitlab-ci.yml file and describe the whole pipeline flow with the jobs. The use of continuous integration now will be automatically and from this stage it is up to the user to use it regarding on the individual needs of each project.

Chapter 5. Frequently Asked Questions

The majority of the issues that someone can face when using the CI infrastructure in Gitlab are presented below.

➔ How to add/give access to another user, in the virtual machine

At some point, it would be useful to let someone from the team to access to a virtual machine that is created by someone else from the team. Below, it is described how to achieve that. For giving a clearer example, assume user A is the owner/creator of the virtual machine and user B is the one who the user A will provide the access rights to the server. From the user's A side, these are the actions that needs to be done:

- `dnf install locmap-release`
- `dnf install locmap`
- `locmap --enable afs`
- `locmap --enable kerberos`
- `locmap --configure all`
- `locmap --enable sudo`
- `addusercern <user's B cern name>`
- `usermod -a -G wheel <user's B cern name>`

User B then from his side, should provide the *id_rsa.pub* key, that he has in lxplus/lxplus8, and then user A copy that in the VM's file: */root/.ssh/authorized_keys*.

After these actions, user A and user B can both access this virtual machine.

➔ In case of an error in the generation, or in the modification of a docker image

In order to enter in the docker image, this command can be used:

```
docker run -it (-e <we can put arguments and some script if we want, like in the entrypoint.sh for example>) --name <docker image new name> <IMAGE ID > bash
```

From the command `docker ps -a` the opened containers can be viewed. Then, in order to commit any change we simply run:

```
docker commit --change='CMD <the command(s) we want to run>' <CONTAINER ID> <docker image new name>.
```

After any commit, the image name changes. So in order to not have many versions of the docker image and always use the latest, the old one has to be deleted and change the name of the newest one. This can be achieved with:

```
docker tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]
```

Otherwise, every time the name of the docker image should be changed in config.toml file, which is not a convenient working approach.

➔ In case that gitlab configuration is needed

At some projects, it may be needed to specify the git configuration, by adding the global user name and user mail. As mentioned before, the docker commit command can be used, but the committed command should be:

```
git config --global user.name CI && git config --global user.email ci@ci.com
```

and move forward to change the tag, once it is finished.

➔ The job or the pipeline is stuck

It can happen sometimes, that the triggered job will not start immediately. The user in that case, has two options. One of the options is to wait for a few minutes and in parallel check in the:

seetings > CI/CD > Runners (collapse the opened tab) > Runners activated for this project

if the specific runner which is responsible to run the job is activated. If it is not, then activate it and refresh the CI/CD > pipelines window.

The second option is to access the server machine. Once you enter the *cispace.cern.ch* these are the commands that you can run in order to start the continuous integration process:

```
sudo gitlab-runner restart
```

```
sudo gitlab-runner verify
```

Once it is finished, you can see that the pipeline will start running immediately.

➔ Add a file/folder in a docker image

Once the docker image has been created, it may needed to add some extra files, or an entire folder into the image. Below, there are some steps presented, on how to do it easily. The steps refer to the

process you follow when the docker image is locally stored and not if the docker image has been pulled from another server.

- Be sure that the docker image has been created (type *docker images* in the terminal). After with the *docker ps -a* you can see the list of the containers. In case that there is no container running the specific image, you can use the command: `docker run -it -e CI_PROJECT_URL=ohwr.org -e GITLAB_USER_ID=<user_ID> -e GITLAB_USER_LOGIN=<account_name> <IMAGE_ID> bash`

- Then copy the file/folder into one directory inside the image:

```
docker cp <host/path/to/file> <CONTAINER_NAME>:</path/to/>
```

- Final step is to commit that action:

```
docker commit <CONTAINER_NAME> <IMAGE_NAME>:<IMAGE_TAG>
```

An example of this process and the extra steps needed when the docker image is stored in another server is the following:

- `export IMAGE_URL=example.com/your_image:your_tag`
- `docker pull $IMAGE_URL`
- `docker create --name temp_container $IMAGE_URL`
- `docker cp /host/path/to/file temp_container:/container/path/to`
- `docker commit temp_container $IMAGE_URL`
- `docker push $IMAGE_URL`

➔ How to share gitlab-runners among different project

Since the gitlab-runners that, as described in this guide, are specific and not shared, that means that they can not be shown/used by default from all the projects in *ohwr.org* or *gitlab.cern.ch*. There are some actions that need to be taken, in order to use a gitlab-runner in different projects. For better understanding, an example is being described below.

First of all, a user in order to add a gitlab-runner in a project, must be a maintainer. This is happening, because only the maintainers have the rights to initialize and configure continuous integration in the project. The maintainer, after registered a runner in the virtual machine or in local server, then can enable the runner to the project, as it is shown in Figure 3. One important thing is to go to settings (the pencil in Figure 10) of each runner and un-click the option “When a runner is locked, it cannot be assigned to other projects” and set the “Maximum job timeout”.

Figure 10. Specific runner in Gitlab

Now this runner is registered, configured and ready to be used. One important thing to be mentioned is that in order this runner to be added in other projects as well is the user to be maintainer in the other projects too. This is the only handicap when it comes to specific runners. A possible alteration to this approach is to use shared runners but this has to go through IT, since they are responsible for these kind of actions.

References

1. https://www.ibm.com/in-en/cloud/learn/docker
2. https://www.ibm.com/in-en/cloud/learn/containers
3. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
4. https://docs.gitlab.com/ee/ci/yaml/gitlab_ci_yaml.html
5. https://docs.gitlab.com/ee/ci/yaml/
6. https://docs.gitlab.com/runner/configuration/advanced-configuration.html
7. https://clouddocs.web.cern.ch/index.html