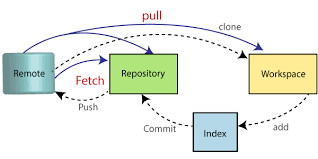
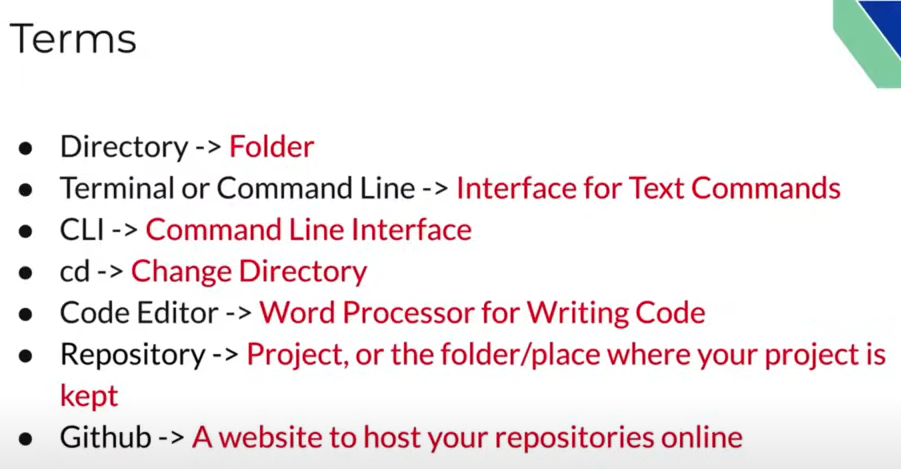
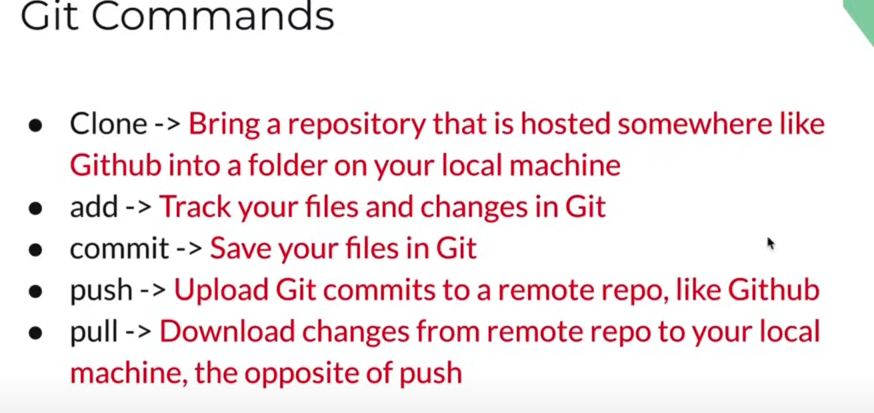
**Gitlab CI/CD**

[**https://nhobethoi.com/lap-trinh/git-cho-nguoi-moi/**](https://nhobethoi.com/lap-trinh/git-cho-nguoi-moi/)

**Git** [**https://youtu.be/RGOj5yH7evk?si=IFNwwhVrtRo619oW**](https://youtu.be/RGOj5yH7evk?si=IFNwwhVrtRo619oW)

* Introduction
  + Git is version control system that can track changes in our project/folder and push&pull changes from remote repositories like GitLab
* 





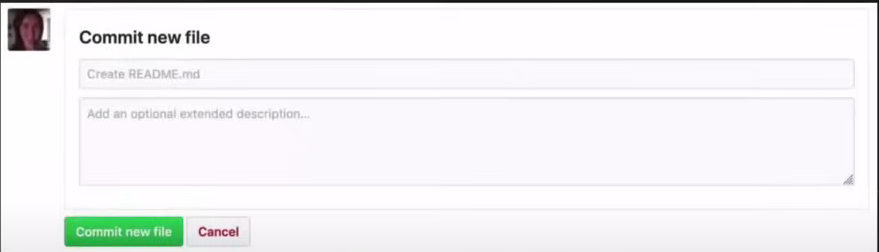
**Step for using git – with exist repo**

Use git clone [git@github.com](mailto:git@github.com)...

Check the changes : git status

Add to commit : git add .

Commit: git commit –m “Message for title” –m “Message for description” (-m means message)

Git commit –am “Use am to add EXIST file that have changes”

Push code: git push origin master : origin is outer directory, master is main branch

!Note use upstream –u : git push -u origin master and next time we just use: git push

**SSH Key**: use this to prove us is the owner of the account

To create key on computer, open terminal: ssh-keygen –t rsa –b 4096 –C “[exactEmailAddress@example.com](mailto:exactEmailAddress@example.com)”

Parameters:

-b "bits" : number of bit in key (more bit means more secure)

-t “Type”: rsa or dsa

-q: ssh-keygen quiet mode.

After doing this, we have 2 file : testkey (private key) and testkey.pub (public key)

## [Adding your SSH key to the ssh-agent](https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent?platform=linux#adding-your-ssh-key-to-the-ssh-agent)

Before adding a new SSH key to the ssh-agent to manage your keys, you should have checked for existing SSH keys and generated a new SSH key.

1. Start the ssh-agent in the background.

$ eval "$(ssh-agent -s)"

> Agent pid 59566

Depending on your environment, you may need to use a different command. For example, you may need to use root access by running sudo -s -H before starting the ssh-agent, or you may need to use exec ssh-agent bash or exec ssh-agent zsh to run the ssh-agent.

1. Add your SSH private key to the ssh-agent.

If you created your key with a different name, or if you are adding an existing key that has a different name, replace id\_ed25519 in the command with the name of your private key file.

ssh-add ~/.ssh/id\_ed25519

1. Add the SSH public key to your account on GitHub. For more information, see "[Adding a new SSH key to your GitHub account](https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account)."

To remove Identity You need to start ssh-agent with this command

eval `ssh-agent -s`

For Remove All Identity

ssh-add -D

To Remove Specific Identity

ssh-add -d ~/.ssh/sshkeynamewithout.pub

For Find a List of available Identify by using this command

ssh-add -l

Before coming to next step, we need to config to let github know who we are:

git config --global user.email "[exactEmailAddress@example.com](mailto:exactEmailAddress@example.com)" (cat ~/.gitconfig to see result)

git config --global --unset user.email : to delete all global user.email

git config --global --remove-section user.email “[exactEmailAddress@example.com](mailto:exactEmailAddress@example.com)” :to delete specific mail

If we only want above email for that folder only, use:

git config user.email “[exactEmailAddress@example.com](mailto:exactEmailAddress@example.com)”

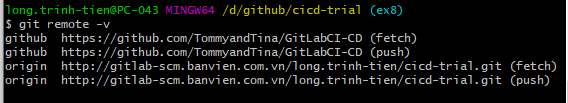
**Step for using git – start locally**

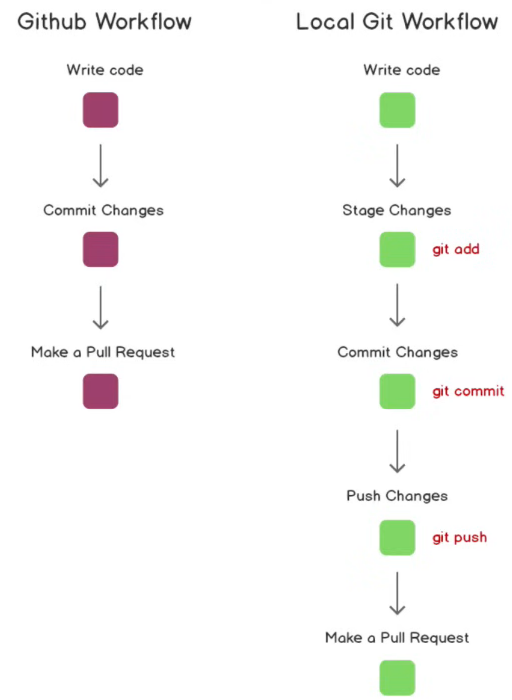
Initialize the repo locally: git init

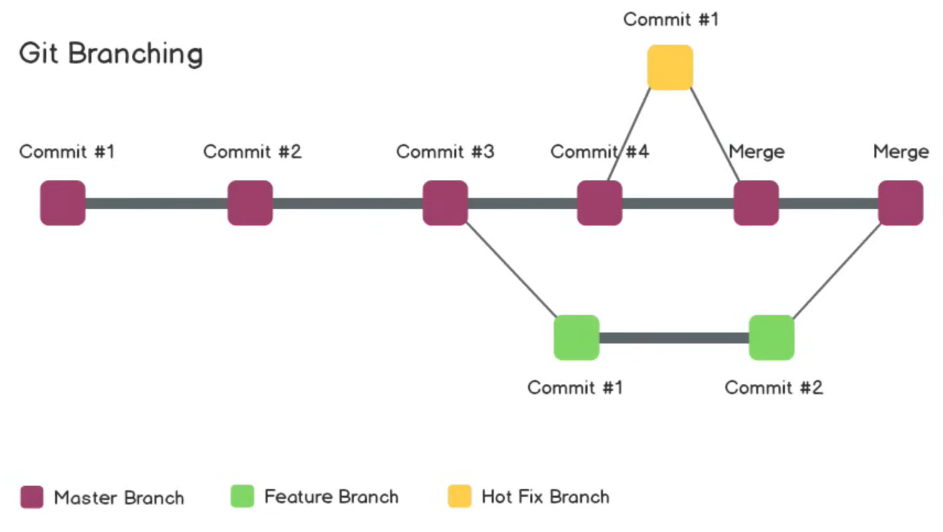
After that, git add . and git commit –m “message”

After push to repo, we need to have a blank repo on github -> git remote add origin git@github... (checking by git remote –v) -> git push

!note : “origin” just is a name, if we add another remote, we need to change origin to another name







* Use ‘ git branch’ to see current available branch
* Use ‘git checkout –b newBranchName’ to create new branch from current branch
* Go to master: git checkout master (in this case we can change master with whatever branch we have, we can use any word inside that branch’s name to auto complete, like we can use ‘ster’ to complete ‘master’ instead of starting with ‘mas’
* Check different between current branch with another branch:

git diff nameOfBranchWeWantToCompareWith (git diff – if we want to compare with last commit)

* Merge branch: git merge nameOfBranch (should use for merge update from main to sub-branch, use Pull Request/Merge Request to merge into main branch)
* Pull Request/Merge Request: we need to do this on graphic interface(on web)
* After that we need to pull that code to local computer: git pull origin main/master
* Finally we delete unused branch that we create before: git branch –d nameOfBranch

**Confict handling**

Step 1: Use git status to check where the conflict is

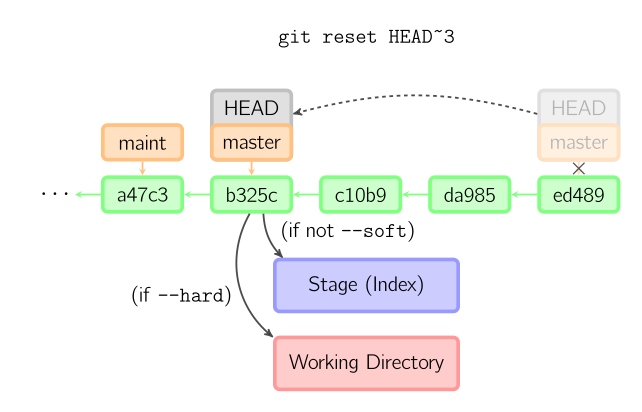
Step 2: Open that file using VSCode, github interface,…the conflict part is inside <<<<<<<, =======, >>>>>>>

Step 3: Handle the conflict (delete 1 , both or keep all of them)

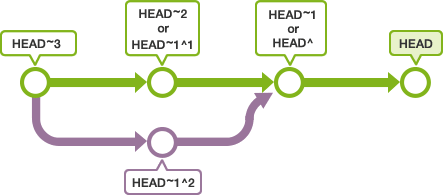
Step 4: use git add conflictFileName to stage changes

Step 5: git commit –m “Resolve conflict”

**Undoing in GIT**



**What are HEAD, Working tree, Index in git?** Git holds a special pointer named HEAD. In Git, this is a pointer to the internal branch you are working on.



On Git, directories placed under Git’s management where people are actually working are called the Working tree.

Between the repository and the working tree exists a place called the index. The index is a place to prepare for committing to the repository.

* Before commit (after add) we can undo the changes: git reset nameOfFile (leave blank name if we want to reset everything)
* After commit (after add) we can undo the changes: git reset HEAD~1 (HEAD is current state, ~num num is number of commits we want to HEAD back)

Git log: see log in your commit history with `HASH`, with that ‘HASH’, we can HEAD to exactly where we want to jump : git reset HASH (this command only delete update that we add to git, if we want to jump back/remove all changes locally, we need to use this command:

Git reset – - hard `HASH`

**git reset –soft B**

git reset --soft B Will return the HEAD to commit B, preserving the changes already made to B in the **Staging Area**. Corresponding to why B we just finished, now we just have a commit like C above.git addgit addgit commit

**git reset B**

git reset B Will return HEAD to commit position B, and also bring changes to **the Working directory**. Corresponding to why B we have just finished coding, not yet.git add

**git reset –hard B**

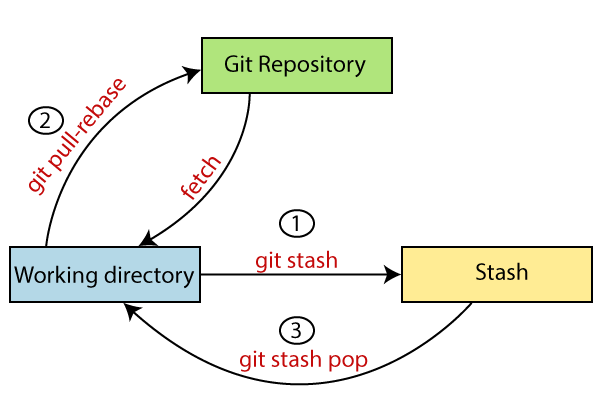
git reset --hard B Will return HEAD to commit position B, and remove all changes at B. Corresponding to at B, we have not coded anything.

**git reset HEAD~1**

Sometimes you see the **command git reset HEAD~1**, it is equivalent to the **command git reset B** as above, because HEAD~1 is backing HEAD back to 1 commit.

* HEAD That's the COMMIT we stand now.
* HEAD~1 is commit immediately before the HEAD on the same branch
* HEAD~n is commit before HEAD n commit on the same branch
* Can be used instead of: @HEADgit log @

**Git stash (happen when changes happen on same files on both local and remote)**



If there are changes that we haven’t merged yet and our local file also have changes, we need to stash (store temporarily) then pull code to local -> apply stash

* First we need to stash: git stash
* After pull code, we can apply: git stash apply
* Or delete stash: git stash pop
* Check our stash lish: git stash list
* <https://nhobethoi.com/git-stash-luu-lai-thay-doi-chua-commit/>

**Change comment of old commit (with last commit we can use git ammend)**

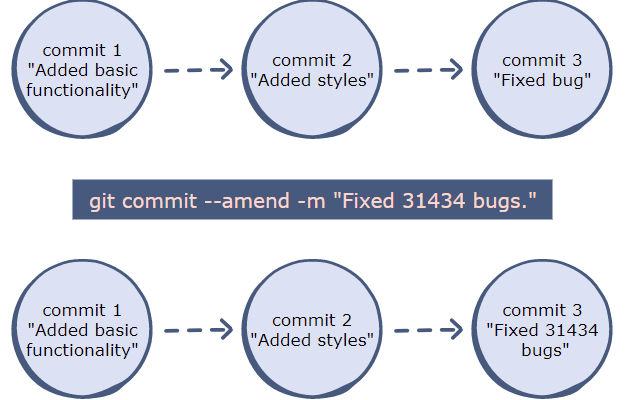


Figure 1: changing last commit

* When we want to change comment, use ‘git log – –oneline’ to check the log we want to change( press q to get out)

git rebase –i HEAD~n /git rebase –i ‘logNum’

or

git rebase – – interactive HEAD~n

After that, we can see a list of commit, at there we replace ‘pick’ with ‘r’ and exit by Ctrl+X, interactive space will appear another window ask us to change the message, we make changes and close window

Finally, we use ‘git push –f’ to force push into remote server

!!note: Collegues need to be informed about this change and make ‘git reset --hard origin/branchWeChangeCommitMessage’

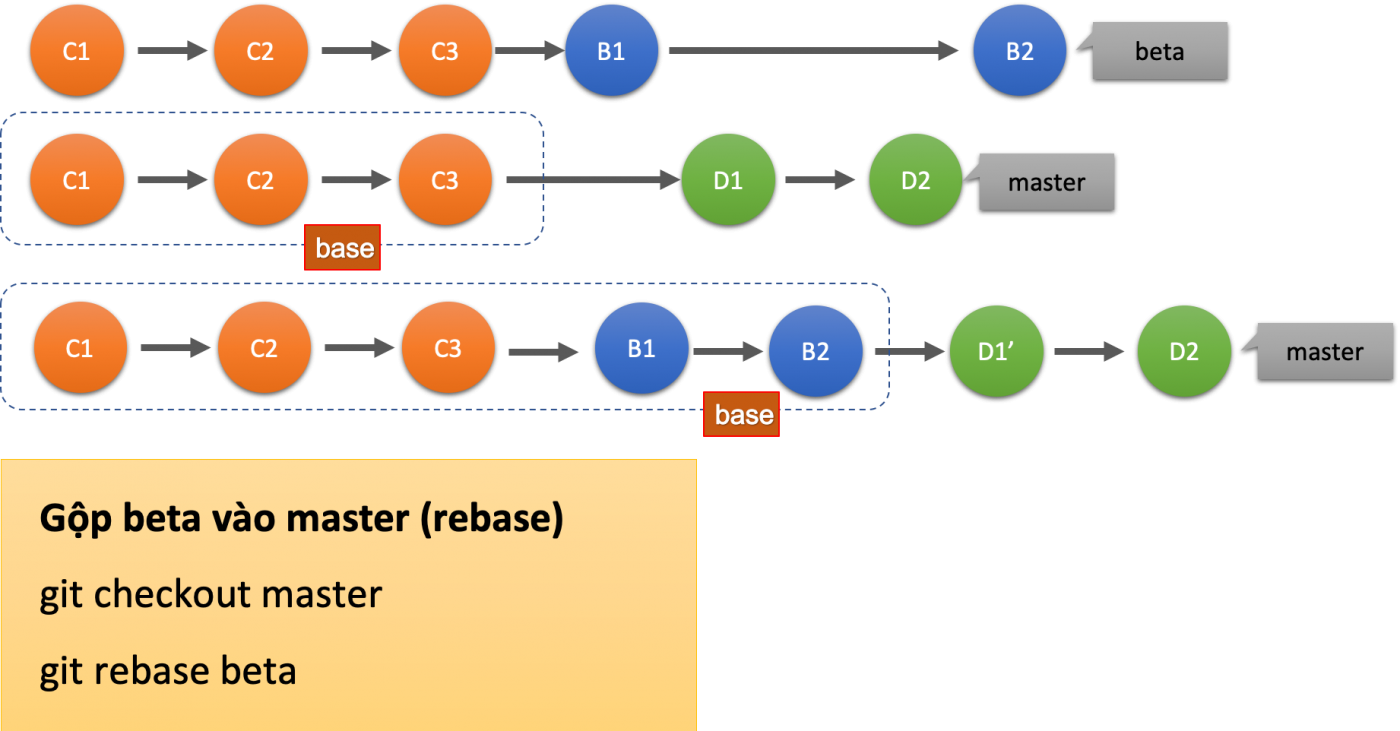
!!! [Never rebase after push](https://www.reddit.com/r/git/comments/135jsrc/never_rebase_after_push_can_someone_explain_that/?rdt=55270).

**How to use rebase**

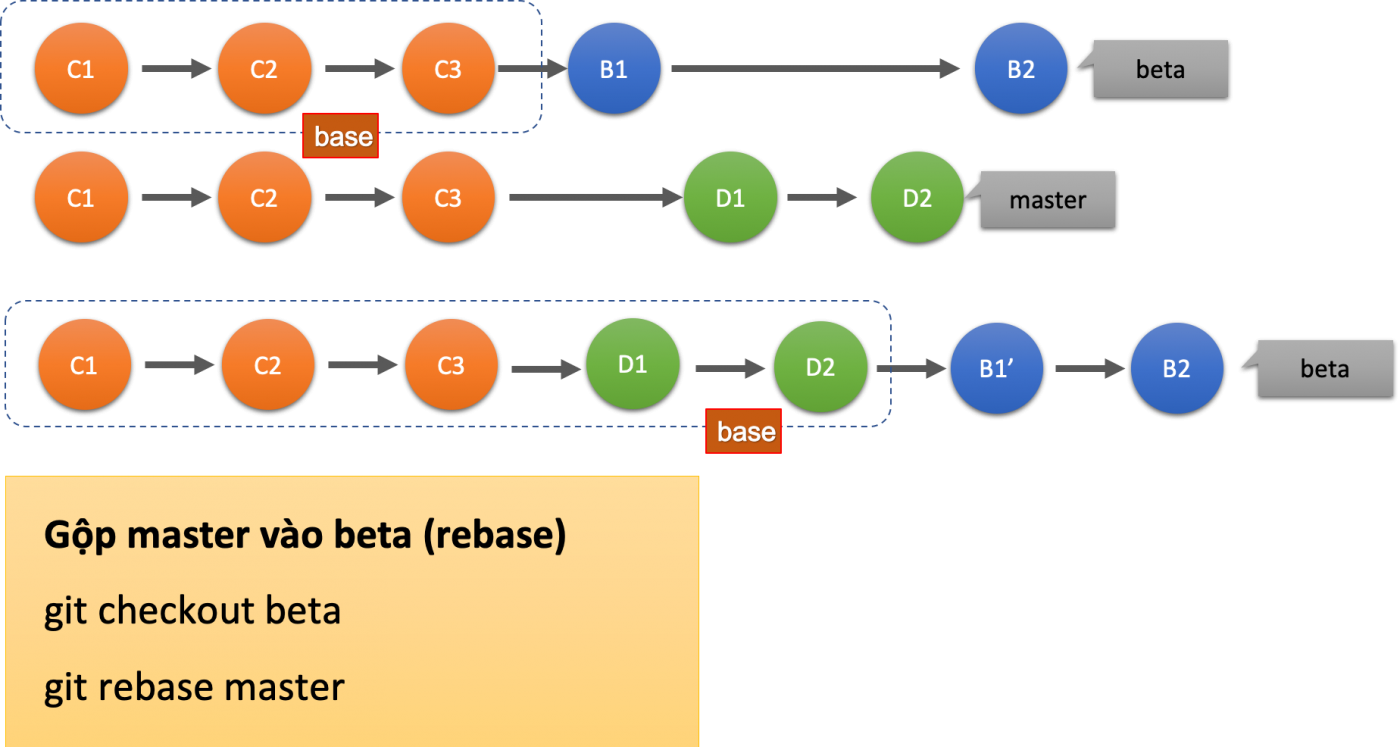
Base means all of previous commit, D1 and D2 in example below are in working space – not commit yet (git add)

Stay in the branch we want to rebase, use:

Git rebase <name\_branch> (<source\_branch> this is optional, useful when we want to make sure the branch we want to rebase)



With above example, we can use ‘git rebase beta master’



Same with above, in this case we use ‘git rebase master beta’

!commit coming from rebase branch will be the lastest commit -> commit history will be changed too

[**Merge commit**](https://hoangphiit.com/post/cach-gop-nhieu-commit-thanh-mot-voi-git-rebase?ref=blog.haposoft.com)

Same step with change comment in commit above, we use rebase interactive to change. For example we have ‘git rebase -i HEAD~3’ with result:

pick 682c63e fix bug login

pick b45bfd8 fix bug login 2

pick 068c3d6 fix bug login 3

We squash 3 commit by:

pick 682c63e fix bug login

s b45bfd8 fix bug login 2

s 068c3d6 fix bug login 3

Save and exit, another window pop up:

# This is a combination of 3 commits.

# This is the 1st commit message:

fix bug login

# This is the commit message #2:

fix bug login 2

# This is the commit message #3:

fix bug login 3

We only keep 1 line for new commit message, leave the rest of commit message stay with #:

# This is a combination of 3 commits.

# This is the 1st commit message:

This is a new commit message!!!

# fix bug login

# This is the commit message #2:

# fix bug login 2

# This is the commit message #3:

# fix bug login 3

Finally, git push -f origin feature to push all that commit online

**Use LF downstream style instead of CRLF in Git**

Expected linebreaks to be 'LF' but found 'CRLF'



To configure git (+repos) ([Check CRLF and LF.md in same folder](https://www.youtube.com/watch?v=zn7m2Mdm_Vg)) or this [link](https://github.com/ascode-com/wiki/blob/main/line-endings/README.md)

* Configure git locally (for all place in system) to use to LF endings:

git config --global core.autocrlf false

git config --global core.eol lf

(To unset these config, git config --global --unset core.autocrlf + git config --global --unset core.eol)

**To Clean all file that is not add or commit (this command erase all file which is different from git reset-only delete from ‘working space’**

git clean -fd

How to push all branch from gitlab to github

Step 1: git clone <URL\_repository\_gitlab>

Step 2:

cd <gitlab\_dir>

git remote add github <URL\_repository\_github>

(!note: on this line we set a new remote name “github” with github url)

Step 3:

git fetch -> git checkout all branch using git checkout <name of branch>

Step 4:

git push github --all

(!note: so we can use the name “github” here)

**CICD**

**CICD process reference**

– Step 1: [Manual] Create the repository and have the default branch of master and dev. Install on Gitlab 9.

– Step 2: [Manual] Except for the owner, coders will push the feature code to the branch dev

– Step 3: [Auto] The system automatically performs test source code, if PASS, it will automatically deploy (rsync) the code to the beta server.

– Step 4: [Manual] Tester / QA will enter the beta system to do UAT (User Acceptance Testing) and confirm everything is OK.

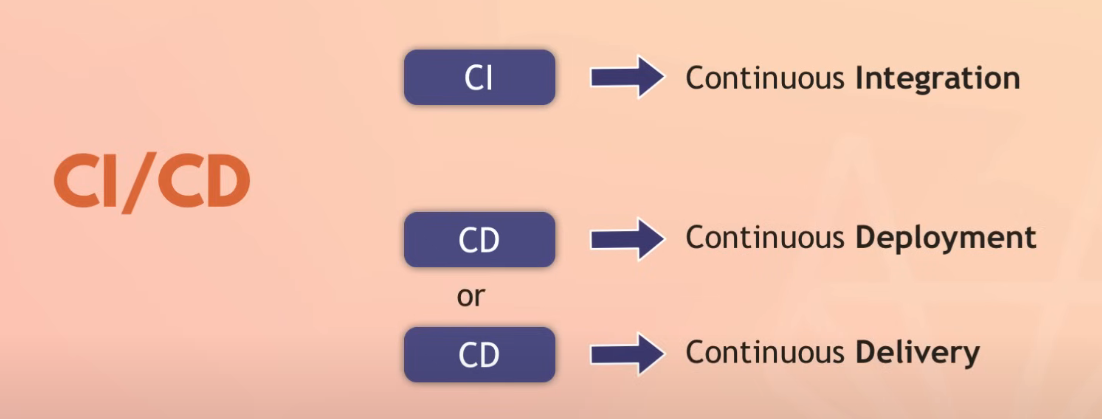
– Step 5: [Manual] Coder or owner will create Merge Request, and merge from branch dev to branch master.

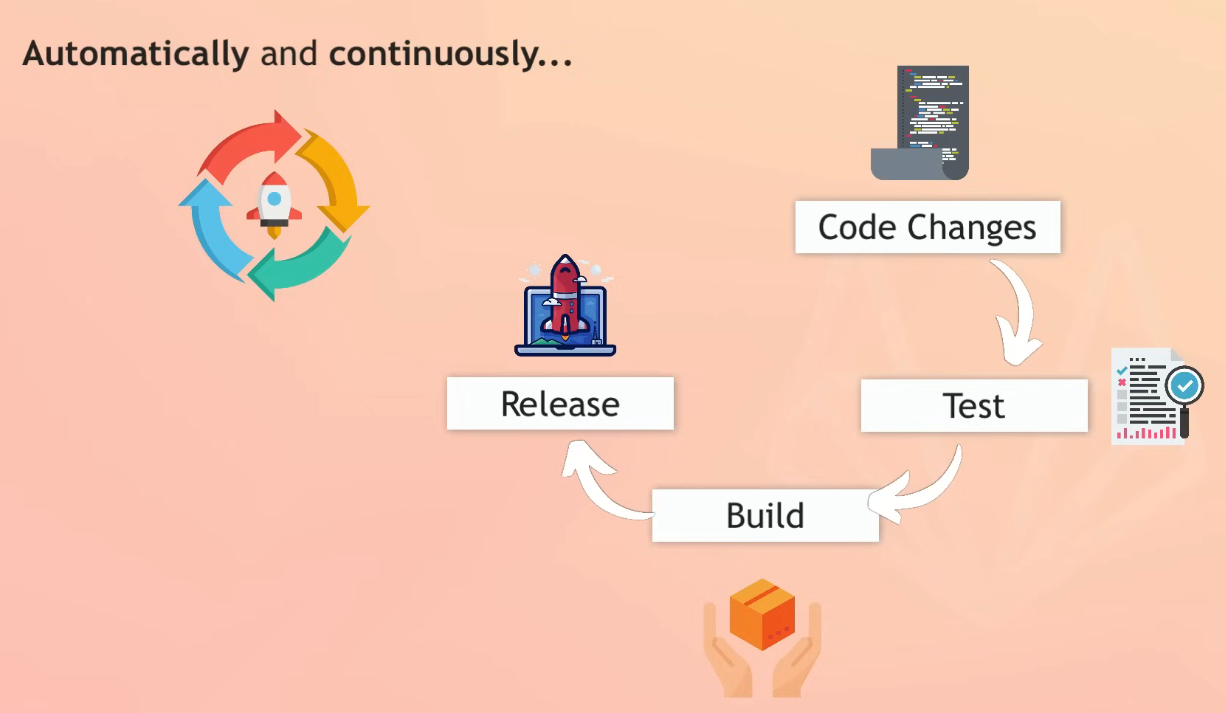
Step 6: [Manual] Owner will accept merge request.

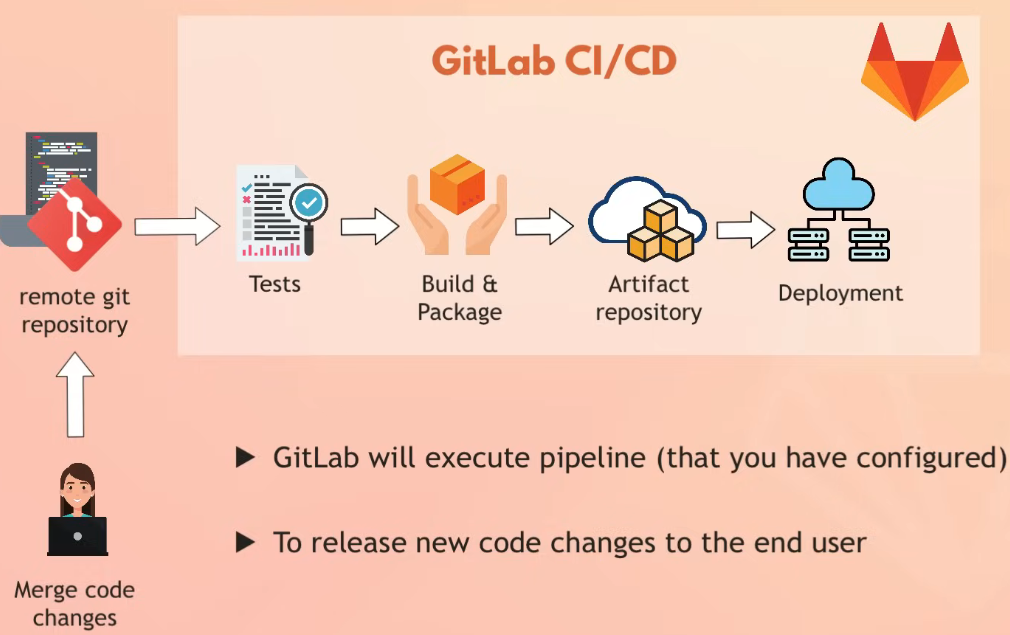
– Step 7: [Auto] The system will automatically execute test source code, if PASS will enable the feature that allows deployment to production server.

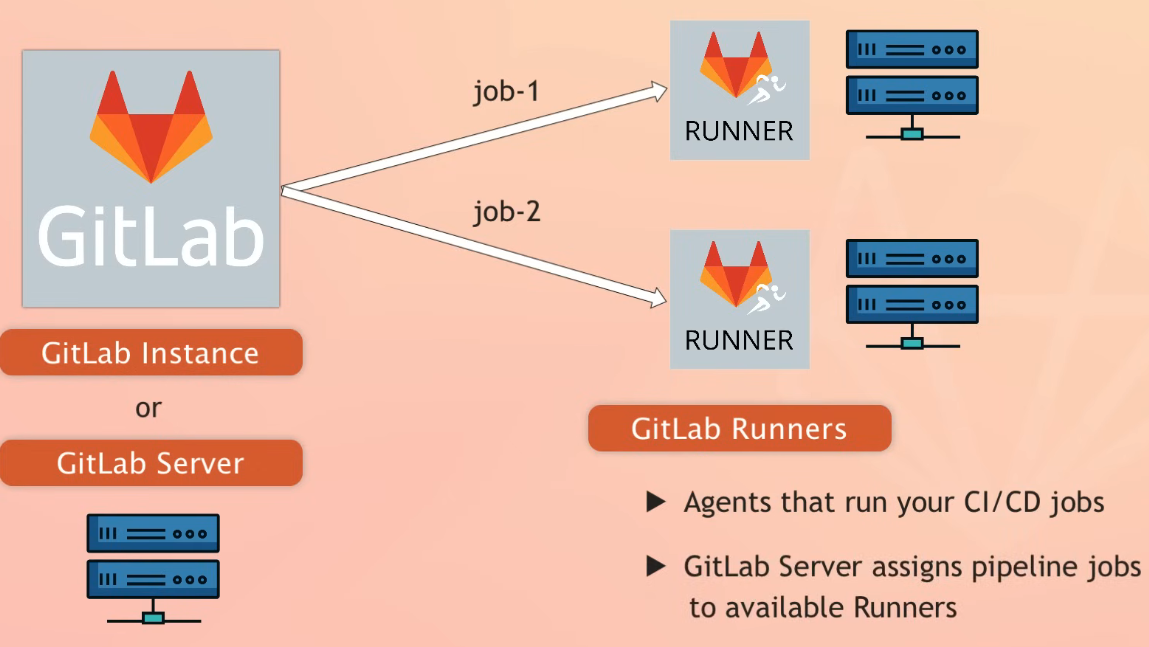
– Step 8: [Manual] Owner review is merge request OK, test OK. Proceed to press the button to deploy the changes to the production environment.

– Step 9: [Manual] Tester / QA will enter the production system to do UAT and confirm everything OK. If not OK, the owner can press the Deploy button of the previous master version to rollback the system to the previous stable state.

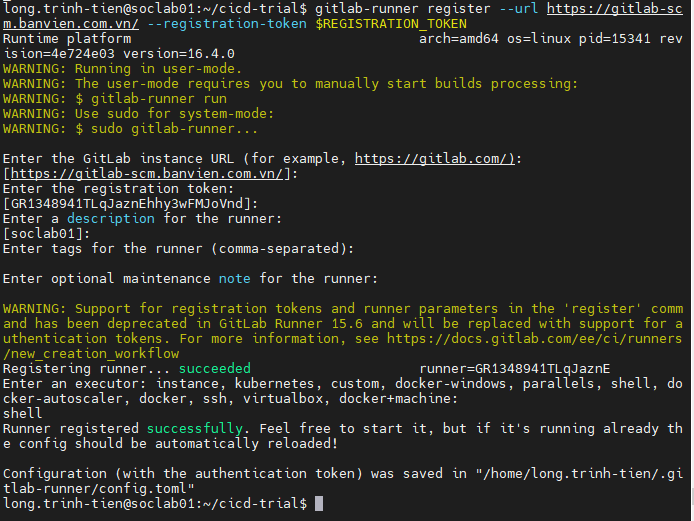








**Gitlab-runner register**

****

!Note: use https instead of http

* After register, we use : ‘gitlab-runner run’ to run runner

**Unregister runnning runner**

**UI way:**

1. Delete runner from UI dashboard:

* open: <https://gitlab.com/gallico/crypto-bot-server/-/settings/ci_cd>
* click on "Runners" section
* click "Remove runner" button in "Available specific runners" section

1. run sudo gitlab-runner verify --delete command, dead runners will be deleted

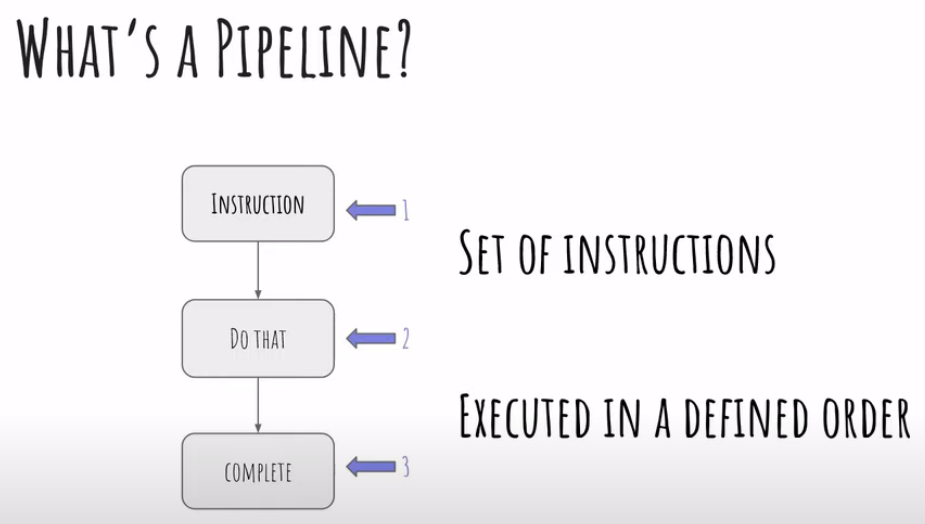
**Bash way:**

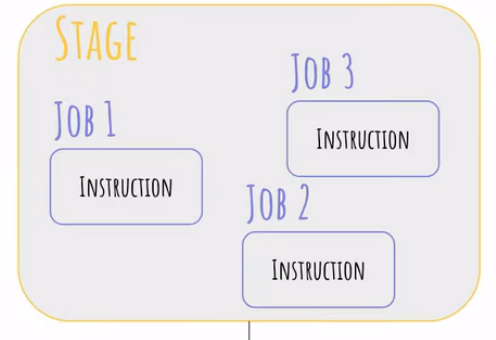
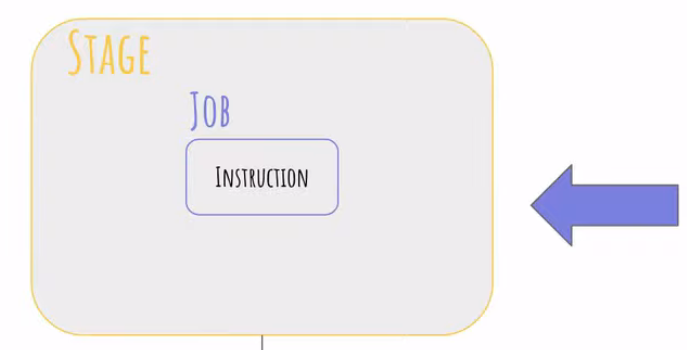
1. SSH to your server
2. If you need to unregister: sudo gitlab-runner unregister --url https://gitlab.com/ --token some-token-goes-here
3. Register new runner: sudo gitlab-runner register --url https://gitlab.com/ --registration-token some-token-goes-here

**Untagged jobs issue**

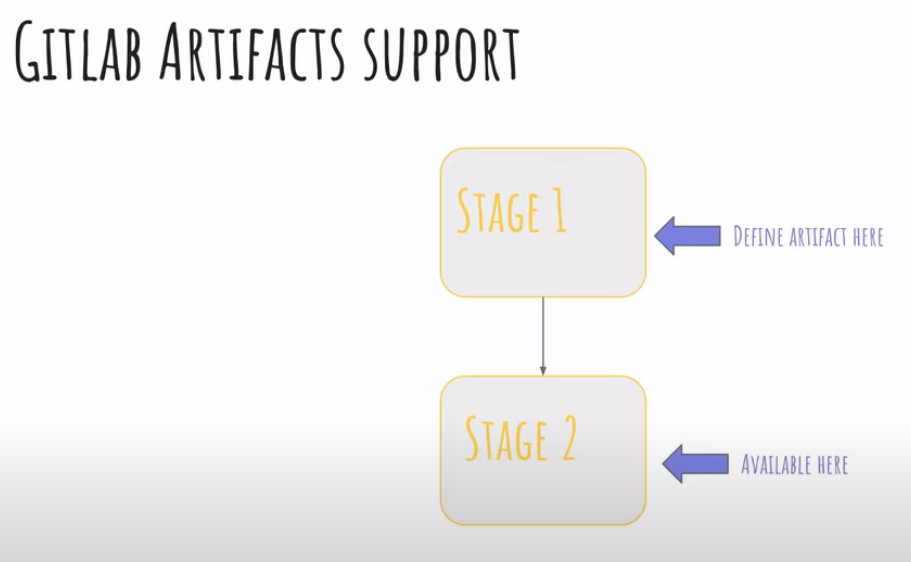
If jobs are stucked you need to enable runner to pick up untagged jobs - it's in Settings -> CI/CD -> Edit runner -> "Run untagged jobs"

**What is pipeline, artifacts, environment?**





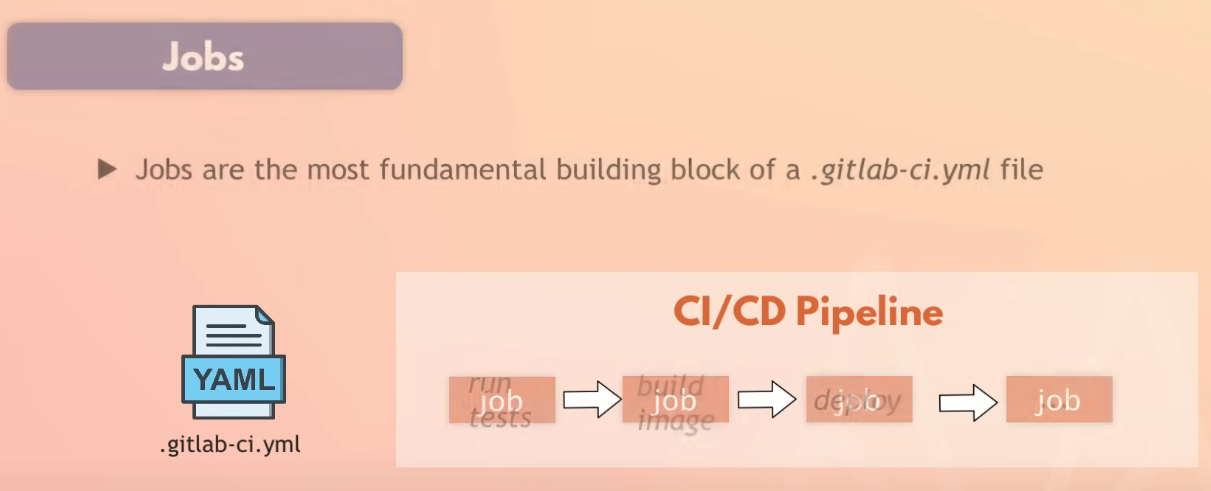
Pipeline includes many stages, every single stage include many jobs

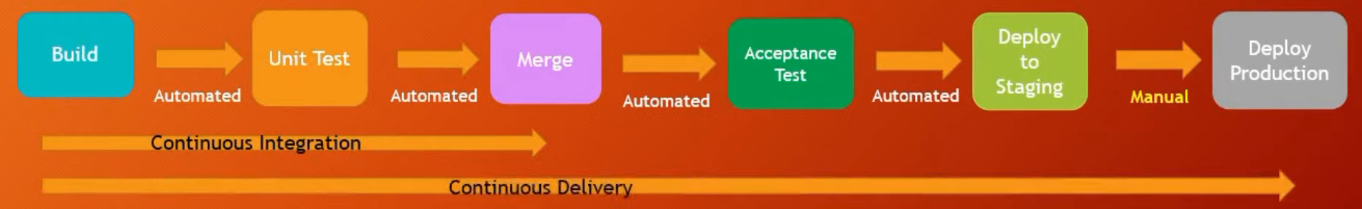


Artifact keep files/folder we want to use for another stage

! Available stages are .pre, build, test, deploy, .post. Another word will lead to error

**Build a job for CI/CD:**



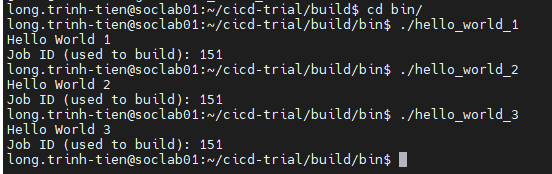
Job are stored in .gitlab-ci.yml. Job will define what to do

CICD process:

* When we commit the code with .gitlab-ci.yml, CICD process will be started
* Gitlab will create a pipeline that contain jobs which are sent to runner, after finishing, they will return the results to Gitlab
* <https://docs.gitlab.com/ee/ci/variables/>

If we have multiple git account on computer

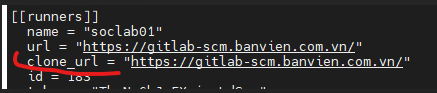
<https://stackoverflow.com/questions/47860772/gitlab-remote-http-basic-access-denied-and-fatal-authentication?page=1&tab=scoredesc#tab-top>

Ex1: 

<job\_id> set ramdomly

Ex2:

Note: add this line to /home/long.trinh-tien/.gitlab-runner/config.toml



* Chmod to 700 for that folder (need permission for us)

Ex3:

How CI/CD works. When a runner starts running, it will perform the following steps:

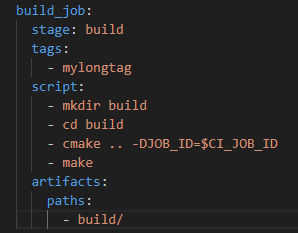
Pull repo: Runner will pull the source code from the repository. It will create a copy (clone) of the repository on the runner server.

Run pipeline: Runner will execute the jobs defined in the pipeline. This can include compiling the source code, running automated tests, and even deploying the source code to a development, test, or production environment.

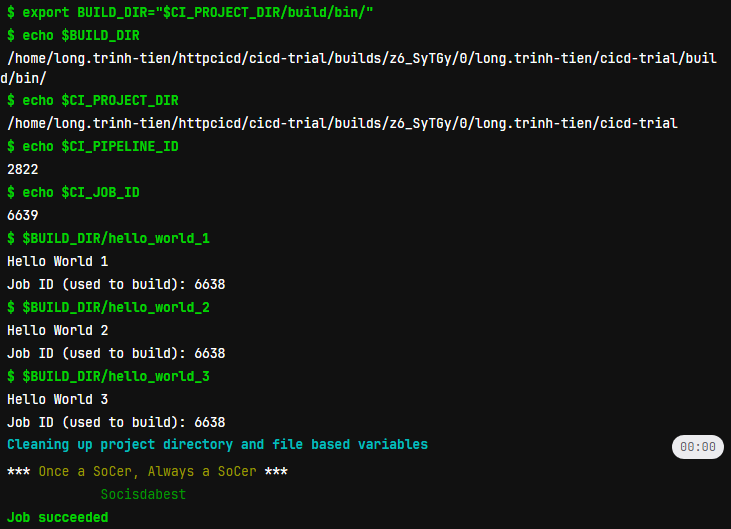
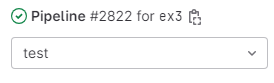
Cleanup: After the pipeline completes, the runner will clean it up and return it to its original state. This means it will delete the copy of the repository it created in step 1.

* We need to add test command into build job. In next part we will use artifact to handle this problem

Ex4

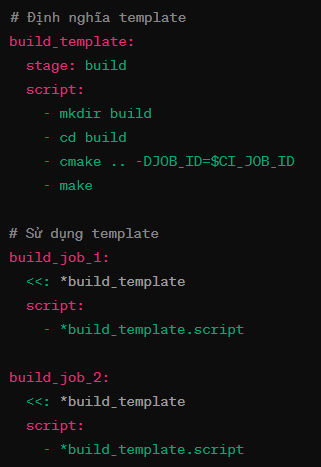
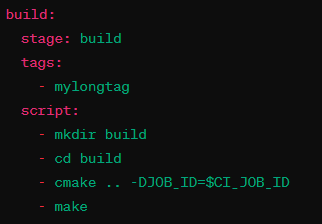
* Need to add artifacts to keep files/folder after completing any other step (Ex4)
* 
* Remember environment variable: $CI\_PROJECT\_DIR is the final direction

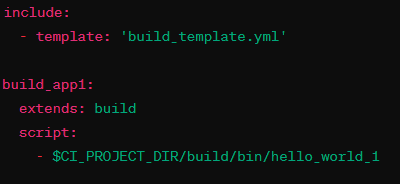
when: manual

* When: manual let us run the test manually
* 
* .

Ex5: template

We have 3 ways:

* First way:
* Second way:
* Third way: -> build\_template.yml

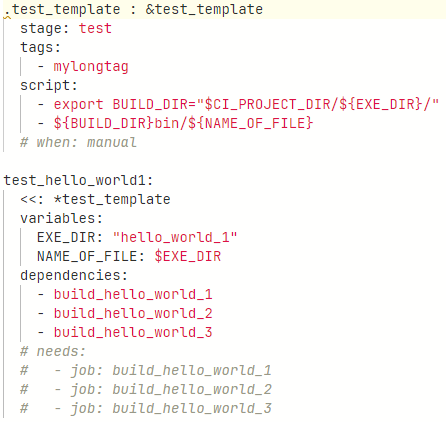


Ex6:

In this exercise, we need to keep result file in build job -> use dependencies/needs to load artifacts from other job

[needs](https://docs.gitlab.com/ee/ci/yaml/#needs) keyword defines real dependencies between jobs, if you define a needs keyword in a job, that job will run only if the needed jobs are successful

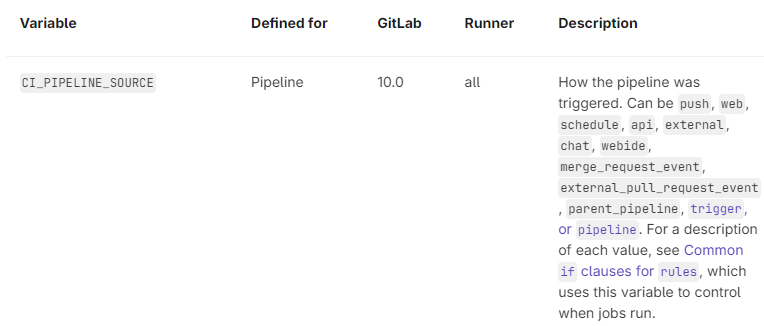
[dependencies](https://docs.gitlab.com/ee/ci/yaml/#dependencies) keyword is used ONLY to define a list of jobs to fetch [artifacts](https://docs.gitlab.com/ee/ci/yaml/#artifacts) from. Otherwise artifacts from all previous jobs are fetched. This is not a hard requirement, so there is no error if the job that you configured in dependencies fails.



Ex7:

Use rules to define how pipeline work. See [predefined variable](https://docs.gitlab.com/ee/ci/variables/predefined_variables.html)





Ex8:

We use exit + error code to make that job jail, allow\_failure : true (accept error, alert by !), false (not allow error)

Job 2 should be ok

job2:

stage: test

script:

- exit 151

allow\_failure: **true**

Job 3 will fail and stop pipeline

job3:

stage: test

script:

- exit 161

allow\_failure: **false**

