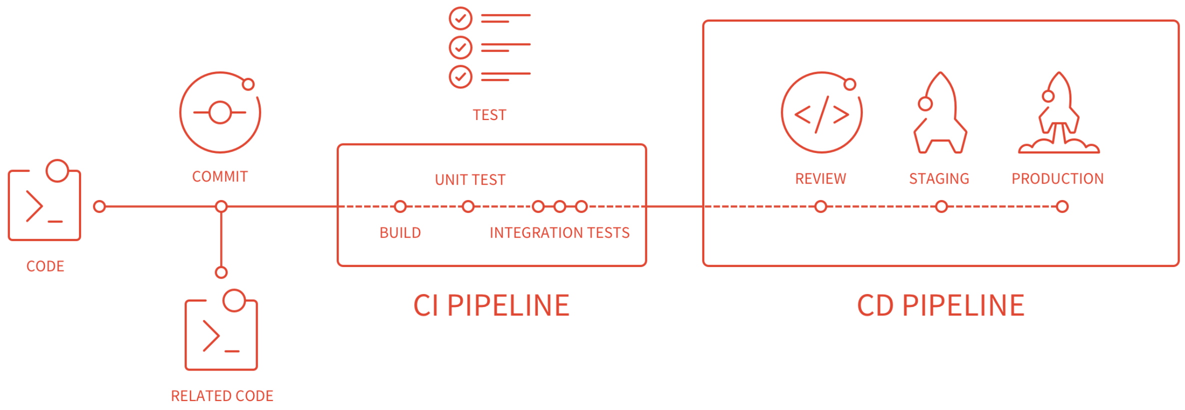
Introduction to Gitlab CI/CD

The benefits of Continuous Integration are huge when automation plays an integral part of your workflow. GitLab comes with built-in Continuous Integration, Continuous Deployment, and Continuous Delivery support to build, test, and deploy your application.

Here's some info we've gathered to get you started.



1. *Gitlab Continuous Integration (CI)*

GitLab offers a continuous integration service. If you add a ***.gitlab-ci.yml file*** to the root directory of your repository, and configure your GitLab project to use a ***Runner,*** then each commit or push triggers your CI ***pipeline.***

The ***.gitlab-ci.yml*** file tells the GitLab runner what to do. By default it runs a pipeline with three stages: ***build***, ***test***, and ***deploy***. You don't need to use all three stages; stages with no jobs are simply ignored.

If everything runs OK (no non-zero return values), you'll get a nice green checkmark associated with the commit. This makes it easy to see whether a commit caused any of the tests to fail before you even look at the code.

Most projects use GitLab's CI service to run the test suite so that developers get immediate feedback if they broke something.

There's a growing trend to use continuous delivery and continuous deployment to automatically deploy tested code to staging and production environments.

So in brief, the steps needed to have a working CI can be summed up to:

1. ***Add* .gitlab-ci.yml *to the root directory of your repository***
2. ***Configure* a Runner**

From there on, on every push to your Git repository, the Runner will automagically start the pipeline and the pipeline will appear under the project's **Pipelines** page.

A. *Creating gitlab-ci.yml file*

Up until now, to set up the build / deploy commands you had to go into GitLab CI and edit the scripts in a form. This made it very low-threshold to set up, but it felt lacking.

We're glad to tell you we'll get a better solution built on the principles and libraries of Travis CI: .gitlab-ci.yml.

What is .gitlab-ci.yml

The `.gitlab-ci.yml` file is where you configure what CI does with your project. It lives in the root of your repository.

On any push to your repository, GitLab will look for the `.gitlab-ci.yml` file and start jobs on *Runners* according to the contents of the file, for that commit.

Because `.gitlab-ci.yml` is in the repository and is version controlled, old versions still build successfully, forks can easily make use of CI, branches can have different pipelines and jobs, and you have a single source of truth for CI. You can read more about the reasons why we are using `.gitlab-ci.yml` in our blog about it.

• *Creating a simple .gitlab-ci.yml file*

You need to create a file named .gitlab-ci.yml in the root directory of your repository. Below is an example for a Ruby on Rails project.

```
before_script:
  - apt-get update -qq && apt-get install -y -qq sqlite3 libsqlite3-dev nodejs
  - ruby -v
  - which ruby
  - gem install bundler --no-document
  - bundle install --jobs $(nproc)  "${FLAGS[@]}"

rspec:
  script:
    - bundle exec rspec

rubocop:
  script:
    - bundle exec rubocop
```

This is the simplest possible configuration that will work for most Ruby applications:

1. Define two jobs rspec and rubocop (the names are arbitrary) with different commands to be executed.
2. Before every job, the commands defined by before_script are executed.

The .gitlab-ci.yml file defines sets of jobs with constraints of how and when they should be run. The jobs are defined as top-level elements with a name (in our case rspec and rubocop) and always have to contain the scriptkeyword. Jobs are used to create jobs, which are then picked by Runners and executed within the environment of the Runner.

What is important is that each job is run independently from each other.

If you want to check whether the .gitlab-ci.yml of your project is valid, there is a Lint tool under the page /ci/lintof your project namespace. You can also find a "CI Lint" button to go to this page under **CI/CD ➔ Pipelines** and**Pipelines ➔ Jobs** in your project.

- ***Push .gitlab-ci.yml to GitLab***

Once you've created .gitlab-ci.yml, you should add it to your Git repository and push it to GitLab.

```
git add .gitlab-ci.yml
git commit -m "Add .gitlab-ci.yml"
git push origin master
```

Now if you go to the **Pipelines** page you will see that the pipeline is pending.
You can also go to the **Commits** page and notice the little pause icon next to the commit SHA. Clicking on it you will be directed to the jobs page for that specific commit.

Notice that there is a pending job which is named after what we wrote in `.gitlab-ci.yml`. "stuck" indicates that there is no Runner configured yet for this job.

The next step is to configure a Runner so that it picks the pending jobs.

B. *Configuring a Runner*

In GitLab, Runners run the jobs that you define in .gitlab-ci.yml. A Runner can be a virtual machine, a VPS, a bare-metal machine, a docker container or even a cluster of containers. GitLab and the Runners communicate through an API, so the only requirement is that the Runner's machine has network access to the GitLab server.

A Runner can be specific to a certain project or serve multiple projects in GitLab. If it serves all projects it's called a *Shared Runner*.

Find more information about different Runners in the Runners documentation.

You can find whether any Runners are assigned to your project by going to **Settings ➔ CI/CD**. Setting up a Runner is easy and straightforward. The official Runner supported by GitLab is written in Go and its documentation can be found at https://docs.gitlab.com/runner/.

In order to have a functional Runner you need to follow two steps:

1. Install it
2. Configure it

Install Gitlab Runner

GitLab Runner can be installed and used on GNU/Linux, macOS, FreeBSD, and Windows. There are three ways to install it. Use Docker, download a binary manually, or use a repository for rpm/deb packages. Below you can find information on the different installation methods.

Here we download binary manually and install the Runner:

To install the Runner:

Add GitLab's official repository:

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

Install the latest version of GitLab Runner, or skip to the next step to install a specific version:

```
sudo apt-get install gitlab-runner
```

After installing the Gitlab Runner we should register the runner:

Registering Runners

Registering a Runner is the process that binds the Runner with a GitLab instance.

To register a Runner under GNU/Linux:

1. Run the following command:

```
sudo gitlab-runner register
```

2. Enter your GitLab instance URL:

Here because the runner and Gitlab instance are in our local machine we type: http://localhost:10080 .

(we can go to the project setting>Ci-CD>Runner. Then we see the address of gitlab ci)

3. Enter the token you obtained to register the Runner:

(we should go to the project setting>Ci-CD>Runner. Then we see the token for that project)

4. Enter a description for the Runner, you can change this later in GitLab's UI:

5. Enter the tags associated with the Runner, you can change this later in GitLab's UI: then at the setting of the runner we can define if the runner just execute the projects with specified tags or it can execute projects without any tag.

6. Enter the Runner executor:

7. If you chose Docker as your executor, you'll be asked for the default image to be used for projects that do not define one in .gitlab-ci.yml: eg: Ubuntu:latest

Now the runner has been successfully registered to the project, if we go to the gitlab instance and check the project we see the "runner", "PipeLine" and the jobs, then we can see the jobs execution by the runner.

Note: ***because the runner and gitlab instance are our local machine, when connecting the runner to the Gitlab Ci it may face an error as follow:***

```
Cloning into '/builds/aaaa/bbbb'...
fatal: unable to access 'http://gitlab-ci-token:xxxxxxxxxxxxxxxxxxxx@localhost:10080/aaaa/bbbb.git/': Failed to connect to localhost port 10080: Connection refused
```

To solving this we can add the following command to the runner's configuration file (you can find at:/etc/gitlab-runner/config.toml).

extra_hosts = ["localhost:172.17.0.1"] *(172.17.0.1 is the address of docker0 bridge, you can see in "ip address")*

Then the conf.toml file will be like this:

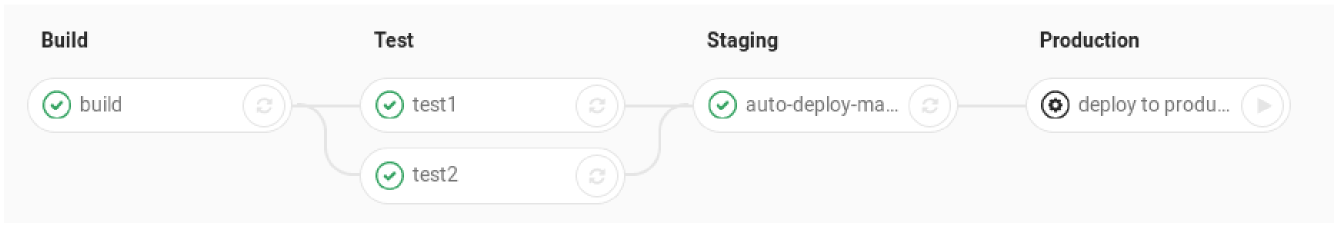
```
concurrent = 1
check_interval = 0
[session_server]
  session_timeout = 1800

[[runners]]
  name = "new-runner"
  url = "http://localhost:10080"
  token = "PnTnqHN_kiySYSgxoRhN"
  executor = "docker"
  [runners.docker]
    tls_verify = false
    image = "ubuntu:latest"
    privileged = false
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    disable_cache = false
    volumes = ["/cache"]
    extra_hosts = ["localhost:172.17.0.1"]
    shm_size = 0
  [runners.cache]
    [runners.cache.s3]
    [runners.cache.gcs]
```

2. *Pipelines*

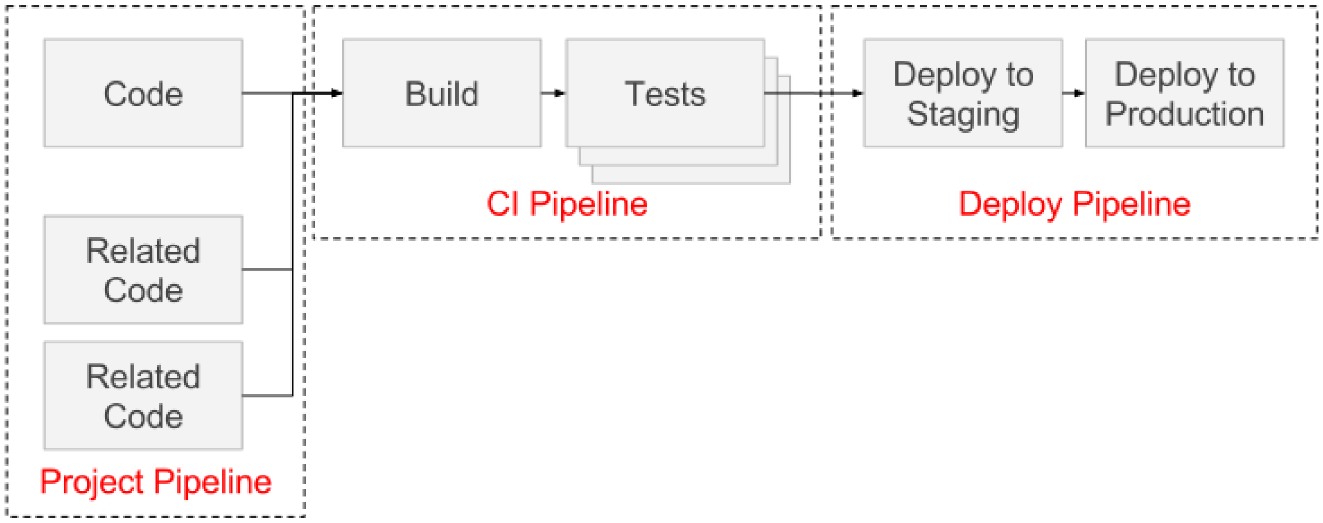
A pipeline is a group of jobs that get executed in stages. All of the jobs in a stage are executed in parallel (if there are enough concurrent Runners), and if they all succeed, the pipeline moves on to the next stage. If one of the jobs fails, the next stage is not (usually) executed. You can access the pipelines page in your project's **Pipelines** tab.

In the following image you can see that the pipeline consists of four stages (`build`, `test`, `staging`, `production`) each one having one or more jobs.



A. *Types of pipelines*

There are three types of pipelines that often use the single shorthand of "pipeline". People often talk about them as if each one is "the" pipeline, but really, they're just pieces of a single, comprehensive pipeline.



1. **CI Pipeline**: Build and test stages defined in .gitlab-ci.yml.
2. **Deploy Pipeline**: Deploy stage(s) defined in .gitlab-ci.yml The flow of deploying code to servers through various stages: e.g. development to staging to production.
3. **Project Pipeline**: Cross-project CI dependencies triggered via API, particularly for micro-services, but also for complicated build dependencies: e.g. api -> front-end, ce/ee -> omnibus.

1. *References*

https://docs.gitlab.com/ee/ci/

https://docs.gitlab.com/ee/ci/quick_start/README.html

https://docs.gitlab.com/runner/install/

https://docs.gitlab.com/runner/register/index.html

https://docs.gitlab.com/ee/ci/pipelines.html