

# DevOps, Docker and Gitlab-CI

## Part 3: Gitlab-CI

Version 1.1.0 (2023-02-15)



-- Cyril zarak Duval, root CRI/ACU 2020

# Gitlab-CI



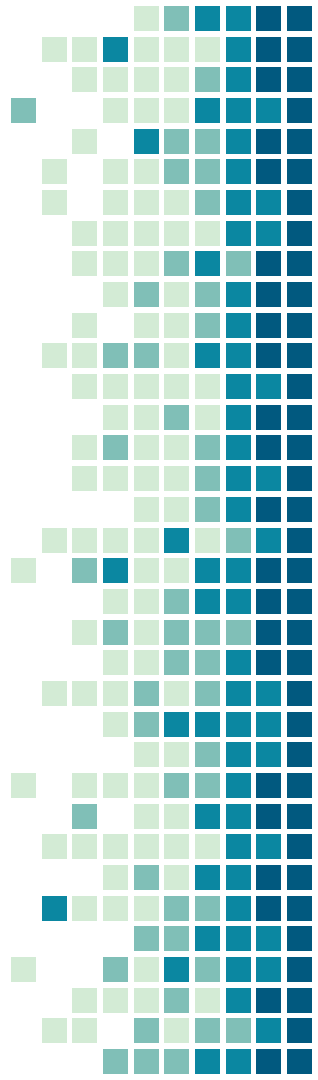
# Git workflow and CI/CD

- You need to use a VCS to work efficiently with many people
  - ◆ Git is obviously the most popular VCS
- The way you work with git is called a git workflow
- A git workflow is a set of rules and best practices for a project or a team
  - ◆ Ex: don't push on master branch directly



# Git workflow and CI/CD

- Usual git workflow looks like this:
  - ◆ master/main/devel branch represents the project “stable” version
    - It’s the most important branch
    - One cannot push directly to it (protected)
    - The ability to merge is limited to maintainers
  - ◆ To add a new feature, one must create a branch



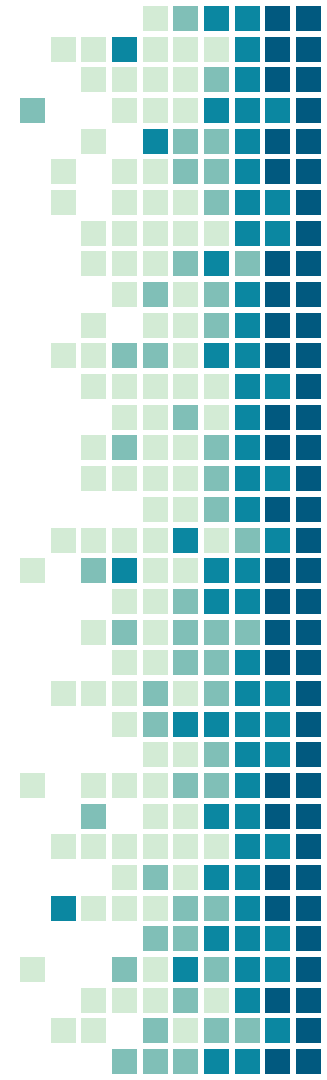
# Git workflow and CI/CD

- Git workflow can also:
  - ◆ Setup a git message format
  - ◆ Allow/force/forbid to squash commits in a branch
  - ◆ Choose between merge commits, fast forward or not, or rebases
  - ◆ Enforce a branch naming convention
  - ◆ Allow or not push force on feature branches
  - ◆ ...



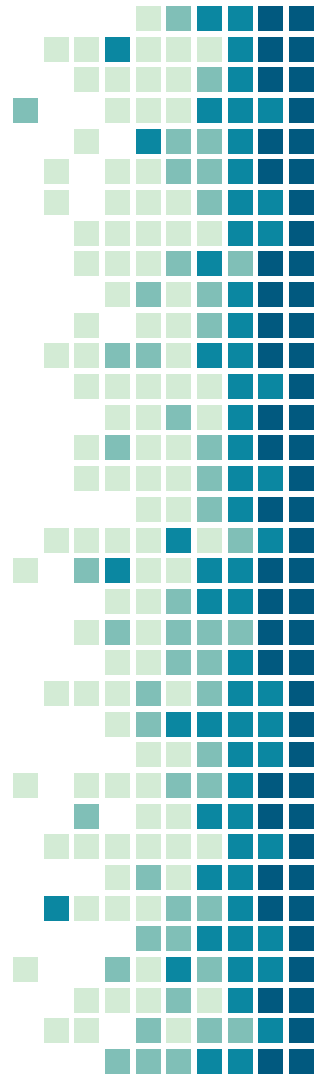
# Git workflow and CI/CD

- Usually you want to assert code quality before merging it
- People review Merge/Pull Requests before merging them
  - ◆ Again, it depends on your git workflow
- People often make mistakes
- Continuous integrations (CI) can help but running your
  - ◆ e2e tests
  - ◆ Unit tests
  - ◆ Linter
  - ◆ ...



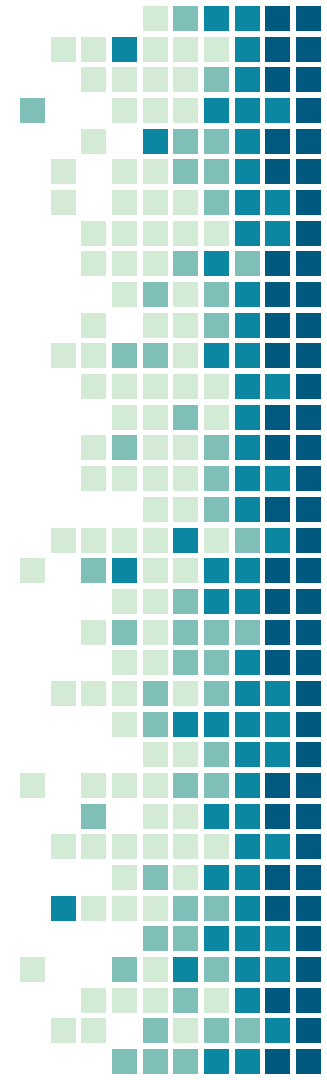
# Git workflow and CI/CD

- CI can also try to compile your project to see if there are errors or warnings
- Make available build on release
- Push the new version somewhere
  - ◆ In this case it's called a CD: continuous deployment
- CI/CD are more or less the same: code to be executed with a git workflow
  - ◆ CI are tests to ensure quality
  - ◆ CD deploys automatically



# Git workflow and CI/CD

- When to run your CI/CD depends on your git workflow
- Examples include:
  - ◆ On each commit
  - ◆ Manually
  - ◆ On tags
  - ◆ On specific branches
  - ◆ Based on the commit name
  - ◆ On merge requests
  - ◆ ...





# Gitlab-CI

- Gitlab comes with a CI/CD system: Gitlab-CI
- In your project, create a `.gitlab-ci.yml`
- This file is a config file describing your CI/CD:
  - ◆ What to do
  - ◆ When to do it
  - ◆ How to do it
- In the project options, CI/CD options available to:
  - ◆ Provide variables (like keys for deployment)
  - ◆ Setup pipeline triggers
  - ◆ ....



# Gitlab-CI and its integration

- Gitlab-ci in itself is a very powerful tool
  - ◆ Checkout the gitlab-ci.yml reference file to be convinced
- Its strength is also with the integration it comes with:
  - ◆ Secrets
  - ◆ Built-in container registry
  - ◆ Specific/shared runners
  - ◆ Badges: pipeline status, coverage, etc
  - ◆ Triggers
  - ◆ Scheduling
  - ◆ Cross-projects



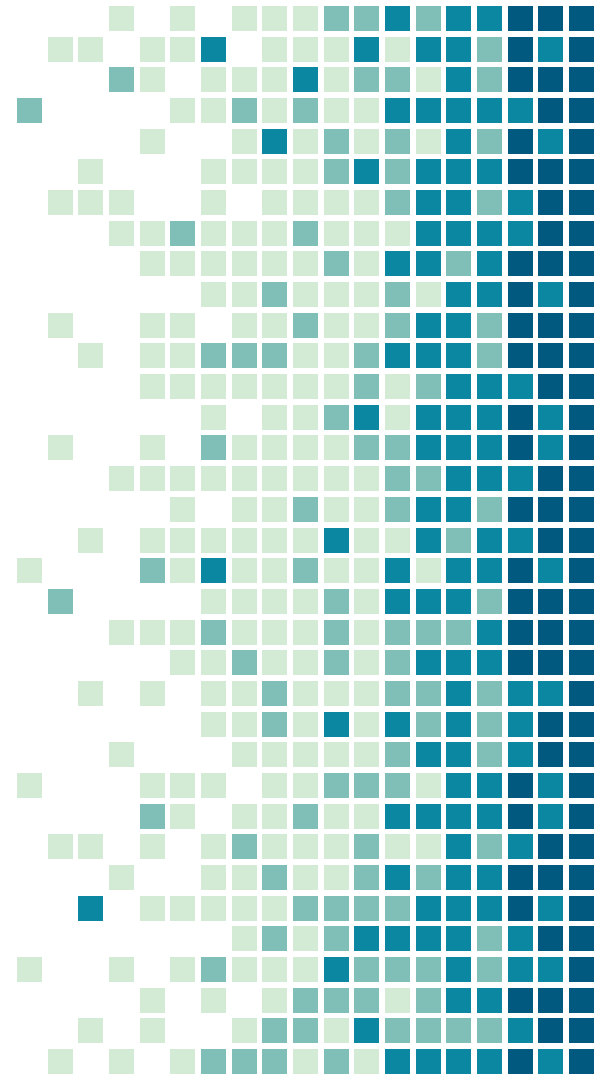
# Gitlab as a DevOps tool

- Gitlab provides services for DevOps in general
- On top of the previously mentioned:
  - ◆ Deployment
    - Tracking deployments
    - Listing platforms with types
    - Well integrated with CI/CD
    - Integration with Sentry
  - ◆ Release
  - ◆ Allow advanced Git workflows
  - ◆ Permission system



# How to write a .gitlab-ci.yml

Don't get confused with jobs, stages, ...



# Gitlab-CI

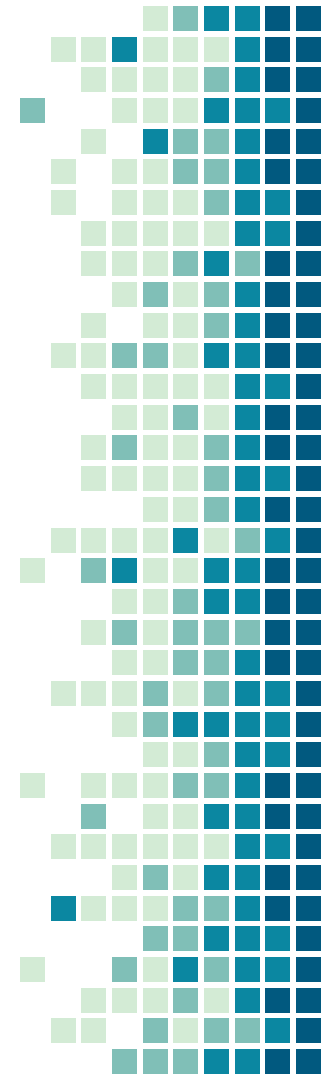
- Gitlab-CI have a concept of pipeline
- A pipeline is a list of stages to execute in a specific order
- A stage is a list of jobs to execute in parallel
- You can put rules for which jobs/stage to run for a specific pipeline
- You can put rules to allow failure in a job to not fail the whole pipeline
- <https://docs.gitlab.com/ee/ci/yaml/> is the reference



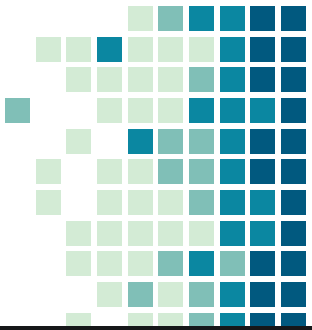
# gitlab-ci.yml

- On the root of the YAML file you can put:
- ◆ A global keyword
  - ◆ A job

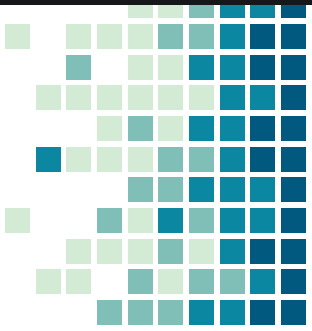
Keyword	Description
<code>default</code>	Custom default values for job keywords.
<code>include</code>	Import configuration from other YAML files.
<code>stages</code>	The names and order of the pipeline stages.
<code>variables</code>	Define CI/CD variables for all job in the pipeline.
<code>workflow</code>	Control what types of pipeline run.



# gitlab-ci.yml



```
1 ---
2
3 stages:
4   - build
5
6 buildWithMake:
7   stage: build
8   script:
9     - make
```



# Gitlab-ci.yml - job

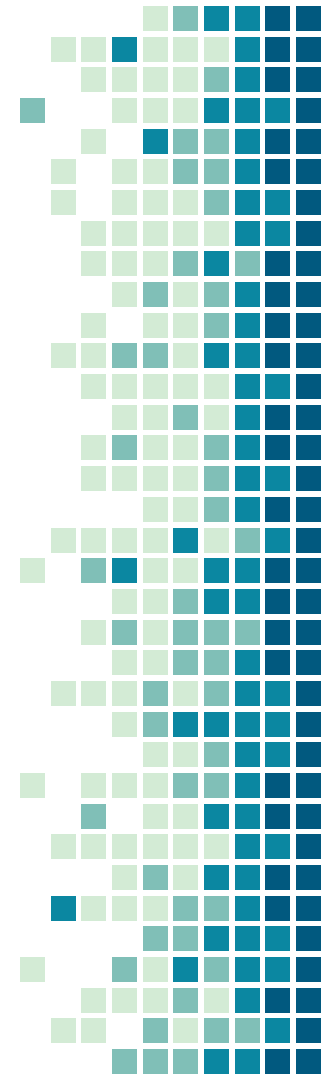
- A job
  - ◆ has a name (its key on the root of the doc)
  - ◆ is in a stage
  - ◆ has a script to run





# Gitlab-ci.yml - job

- Job context is independant
  - ◆ Each job is run in a new environment
    - Except for the artifacts which remain
    - Artifacts are defined explicitly
  - ◆ The script is executed in a directory where the project is
    - It's provided as a git repo, you can do git operations
      - You can even commit from a CI/CD
  - ◆ Environment variables are provided with informations about the job
    - Git commit hash, git branch, repo URL, ...



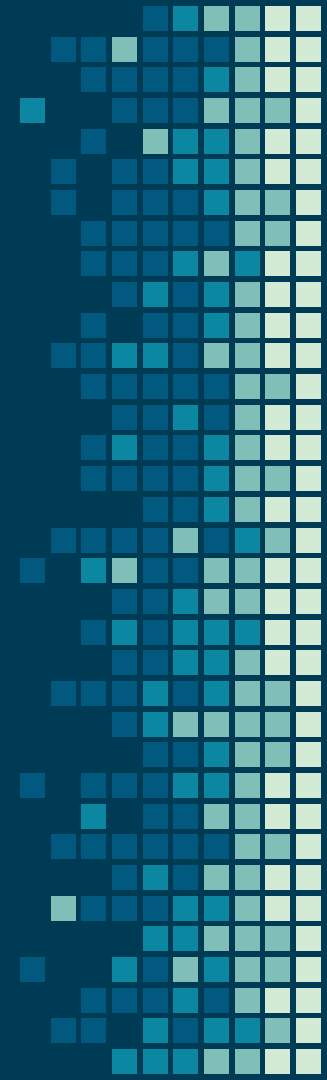
# Gitlab-ci.yml - job

- How can you run a job in a new environment every time ?
  - ◆ Without being able to escape this environment
  - ◆ While being as deterministic as possible
  - ◆ While having a way to choose what will be in the env
    - Like packages, or even the OS
- If you don't have a hint on how to implement that, go back to the first slide
- (sidenote: some people don't use containers as a runner executor. It remains the most popular one though)



# Thanks !

Questions ?



Slides available on [zarak.fr/](http://zarak.fr/)

Contact: [cyril@cri.epita.fr](mailto:cyril@cri.epita.fr)

[zarak production#5492](#)