codurance.com

GitLab CI Training

codurance

Goal

Understand how GitLab Ci pipelines work

What you’ll learn

- Set up a CI pipeline
- Options to configure pipelines
- Why you should do this

What is CI?

Continuous **I**ntegration

Automating the integration of code changes into a single software project.

What is CI?

Continuous **I**ntegration

Automating the integration of code changes into a single software project.

GitLab CI advantages

- Ease of configuration
- Source code security
- Pipeline automation
- GitLab native integration
- Deployment scheduling

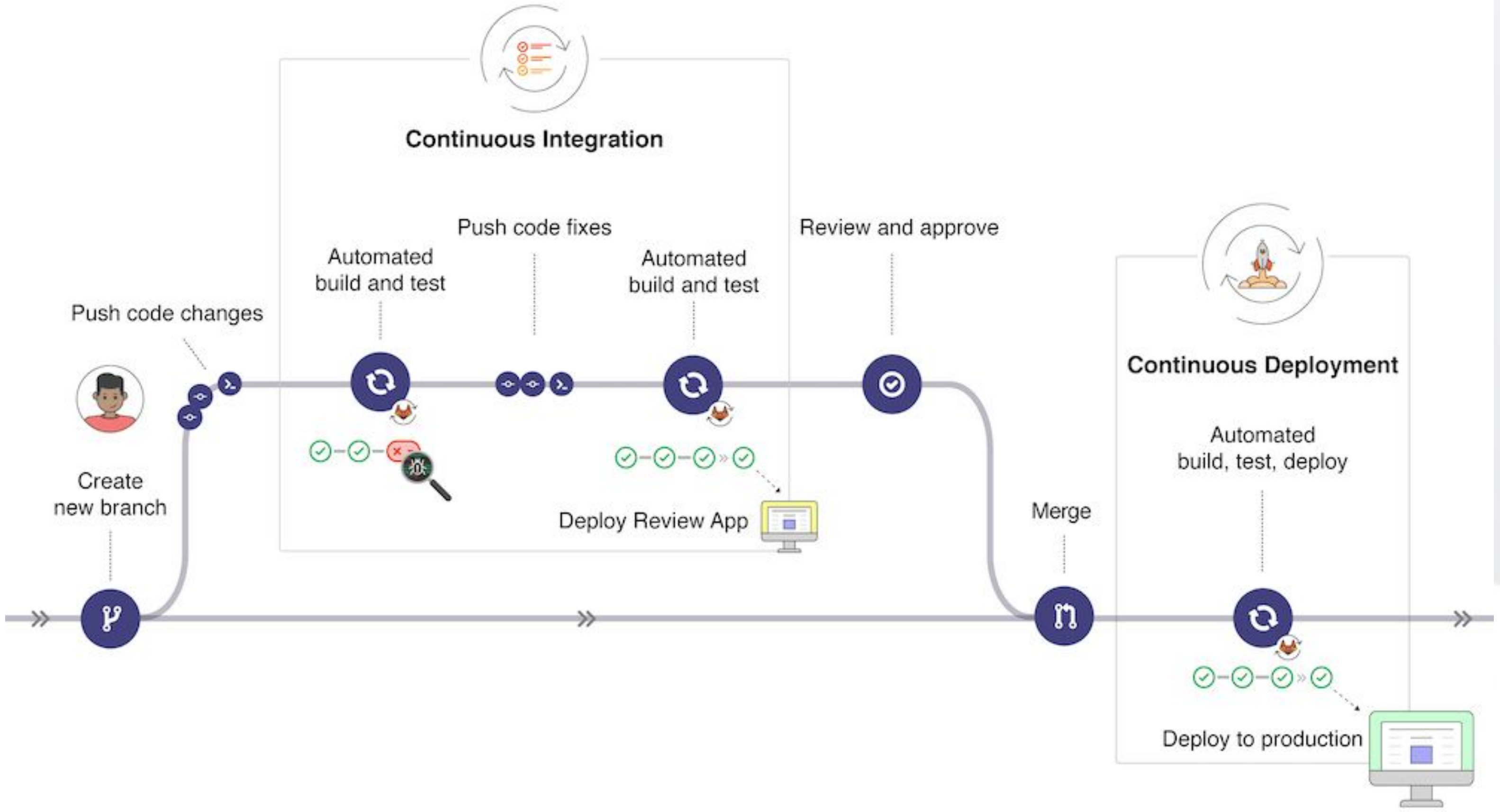
GitLab CI workflow

GitLab CI Workflow

GitLab CI/CD: GitLab continuous methods

- Continuous Integration
- Delivery
- Deployment

GitLab CI Workflow



GitLab CI Training

Pipelines

Pipelines

- A series of automated processes
- Move code updates from VCS to production

Pipelines

Pipelines

Jobs

↓

What?

Stages

Pipelines

Jobs

↓

What?

Stages

↓

When?

Pipelines

Pipelines

1. Build

Pipelines

1. Build
2. Test

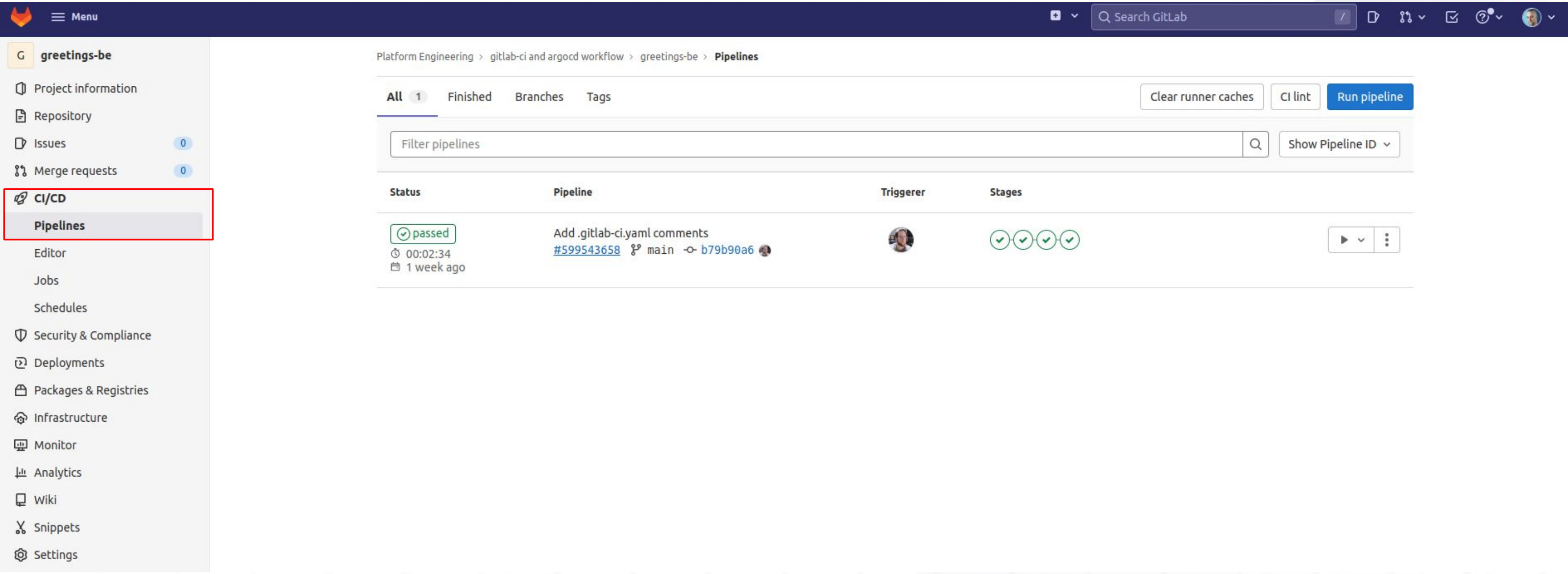
Pipelines

1. Build
2. Test
3. Staging

Pipelines

1. Build
2. Test
3. Staging
4. Production

Pipelines

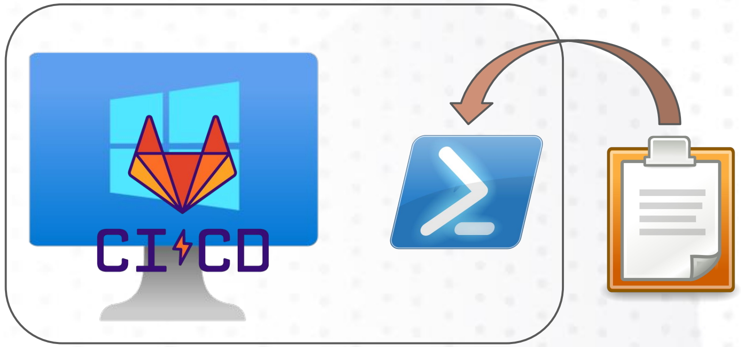


Status	Pipeline	Triggerer	Stages
passed 00:02:34 1 week ago	Add .gitlab-ci.yaml comments #599543658 main b79b90a6		✓ ✓ ✓ ✓

Jobs

- Most fundamental element of a pipeline.
- Defined with constraints.
- Top-level elements with an arbitrary name
- Must contain *at least* the script clause.

Jobs



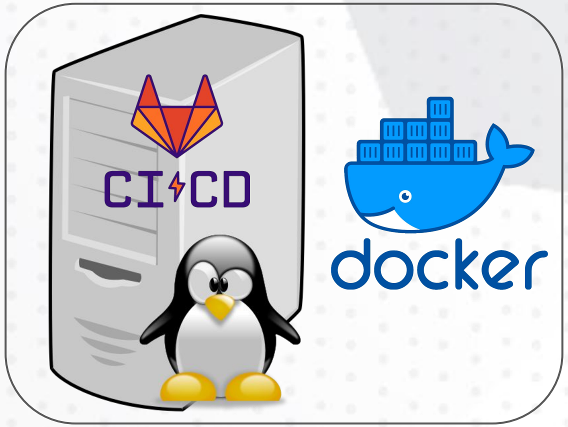
GitLab Runner

An application that works with GitLab CI/CD to run jobs in a pipeline.

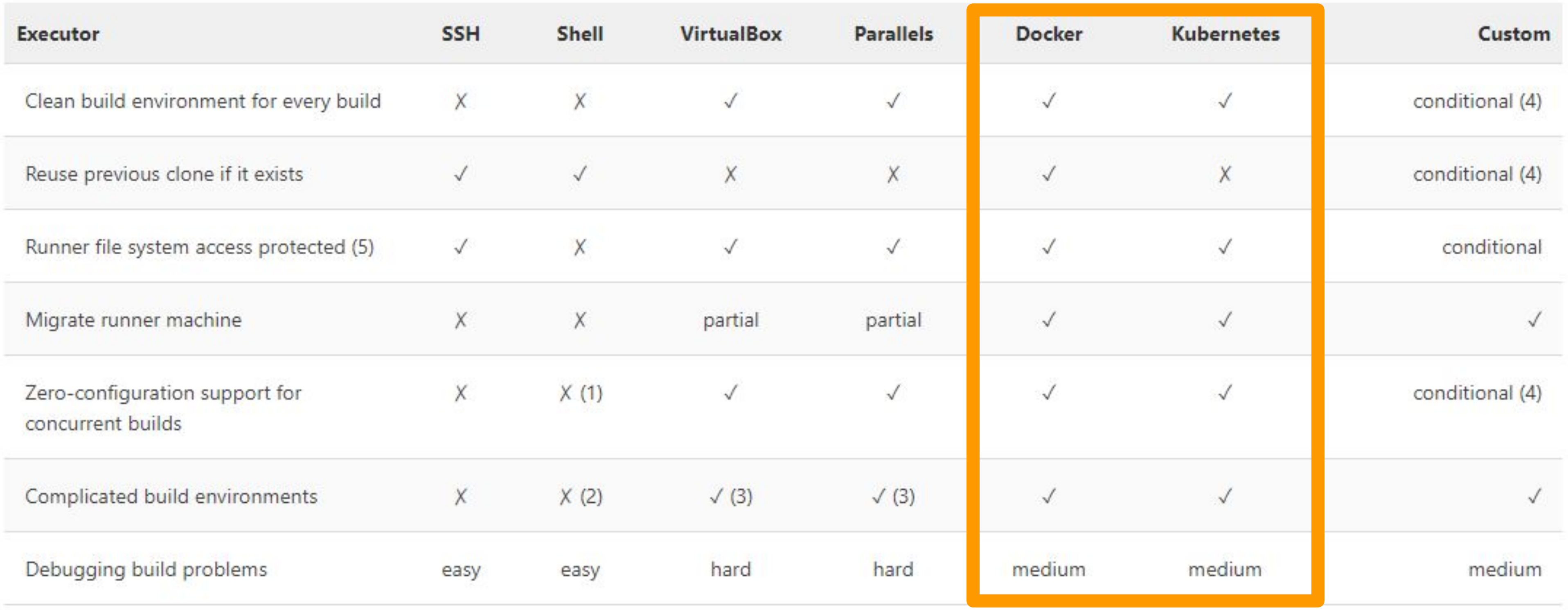
GitLab Executor

When we register a runner, we must choose an *executor*

GitLab Runners + Executors

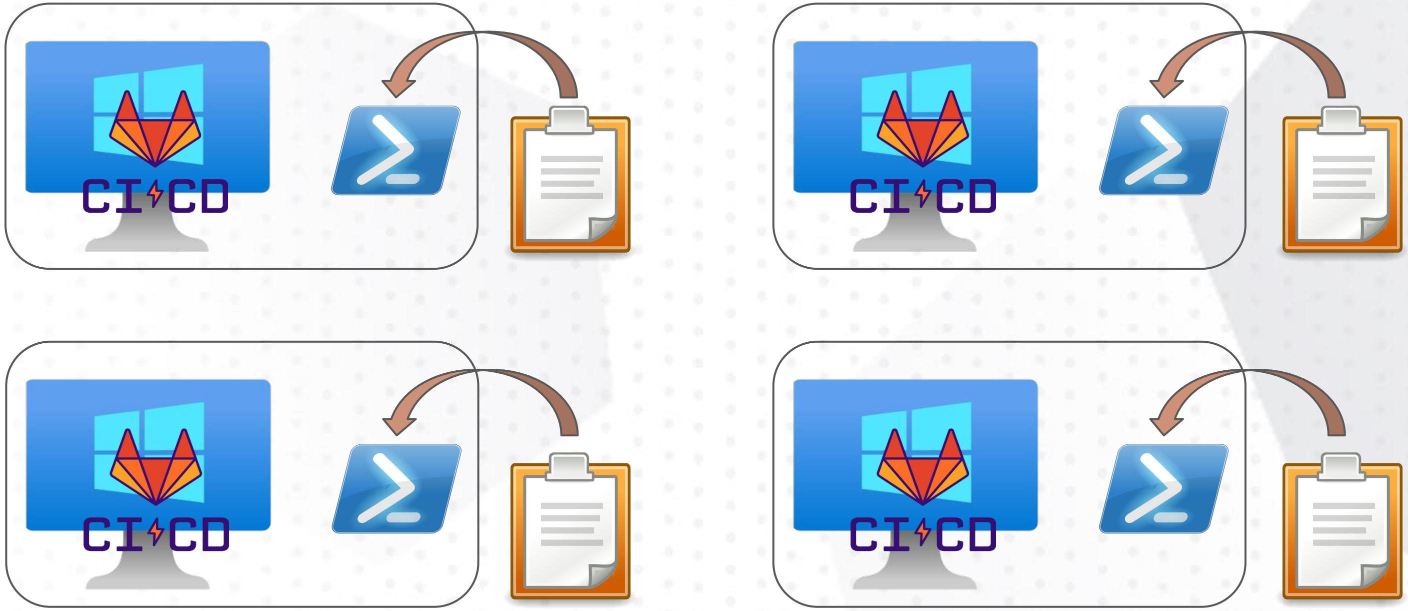


Executors

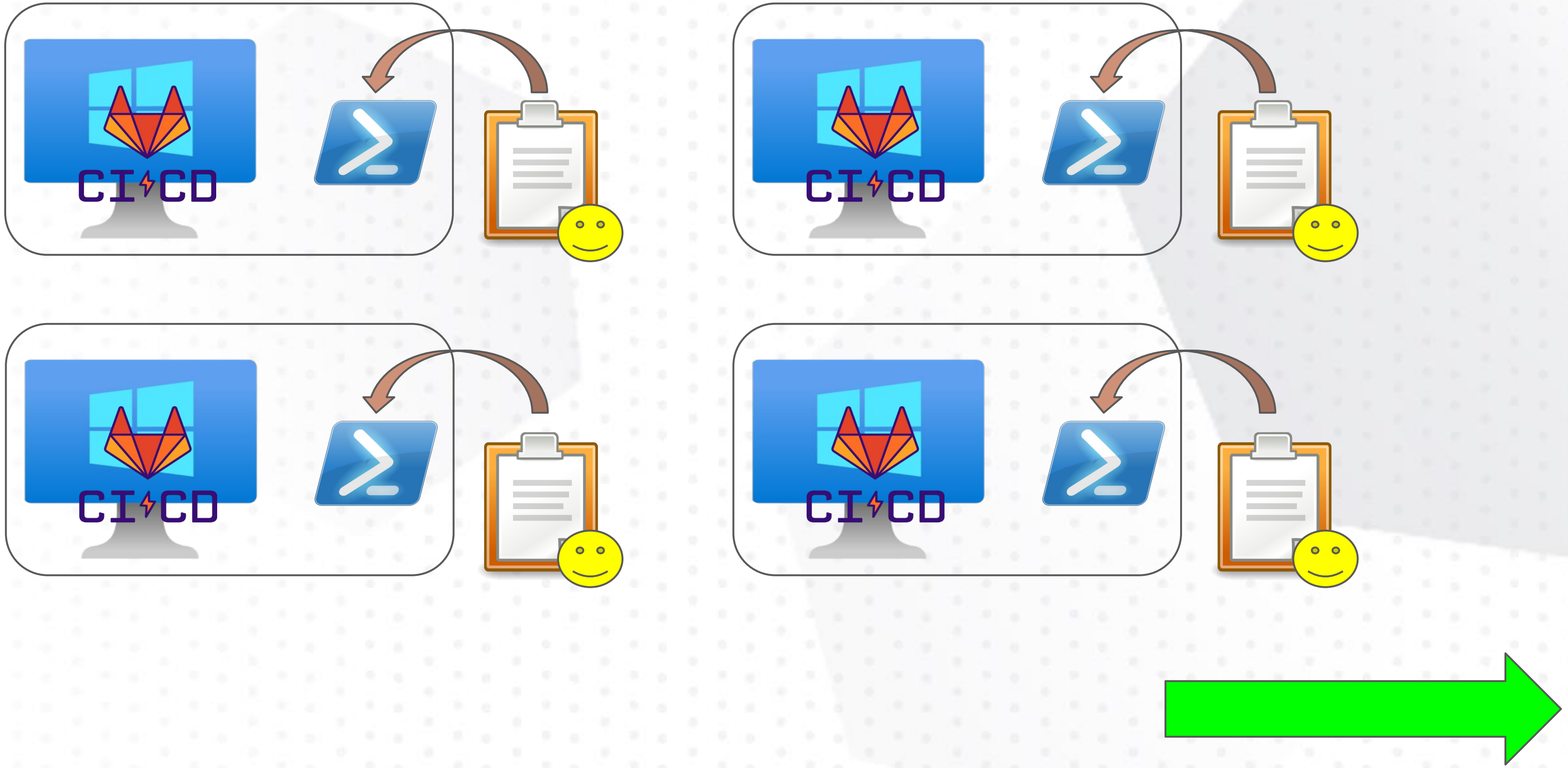


Executor	SSH	Shell	VirtualBox	Parallels	Docker	Kubernetes	Custom
Clean build environment for every build	✗	✗	✓	✓	✓	✓	conditional (4)
Reuse previous clone if it exists	✓	✓	✗	✗	✓	✗	conditional (4)
Runner file system access protected (5)	✓	✗	✓	✓	✓	✓	conditional
Migrate runner machine	✗	✗	partial	partial	✓	✓	✓
Zero-configuration support for concurrent builds	✗	✗ (1)	✓	✓	✓	✓	conditional (4)
Complicated build environments	✗	✗ (2)	✓ (3)	✓ (3)	✓	✓	✓
Debugging build problems	easy	easy	hard	hard	medium	medium	medium

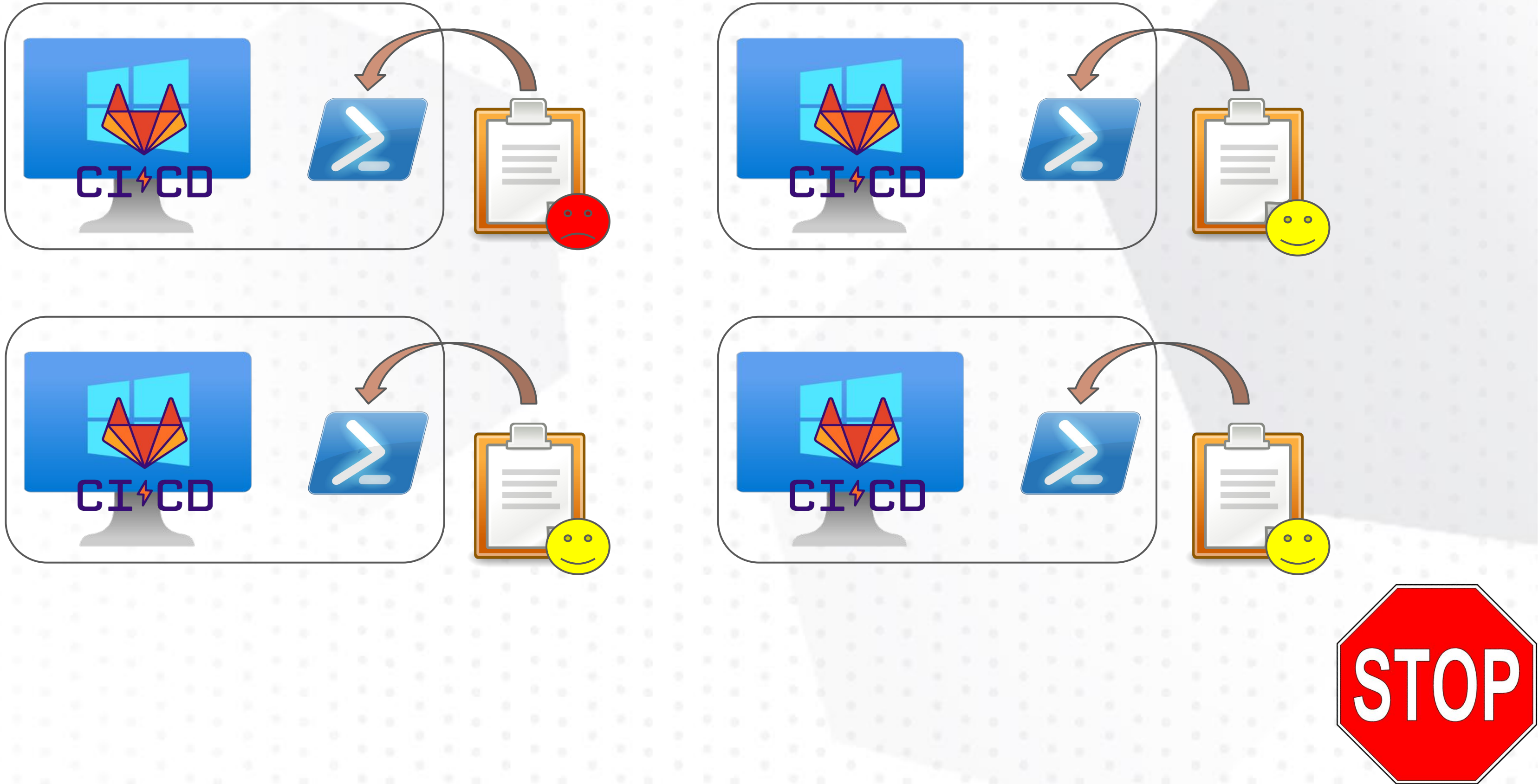
Jobs



Jobs



Jobs



GitLab CI Training

Pipeline Configuration

Pipeline configuration

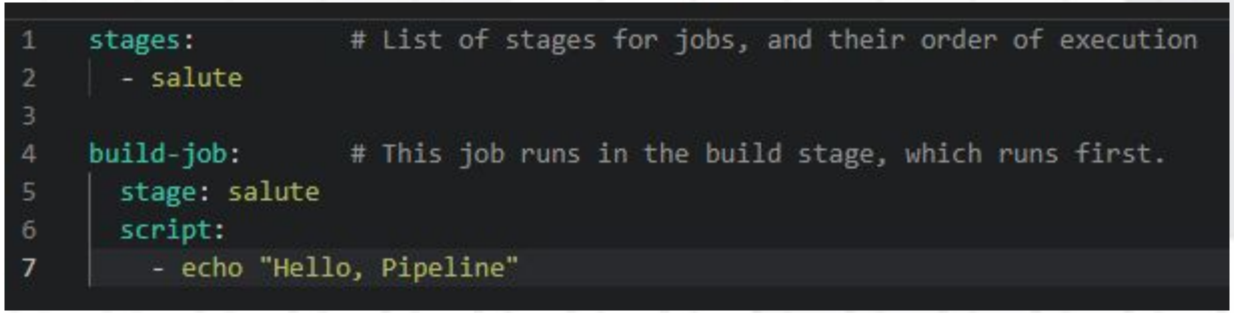
1. Jobs definition

.gitlab-ci.yml

- Define, declaratively, how we want to run our pipeline
- Needed to run the pipelines
- Must be located in the root of the repository

Exercise: My first pipeline

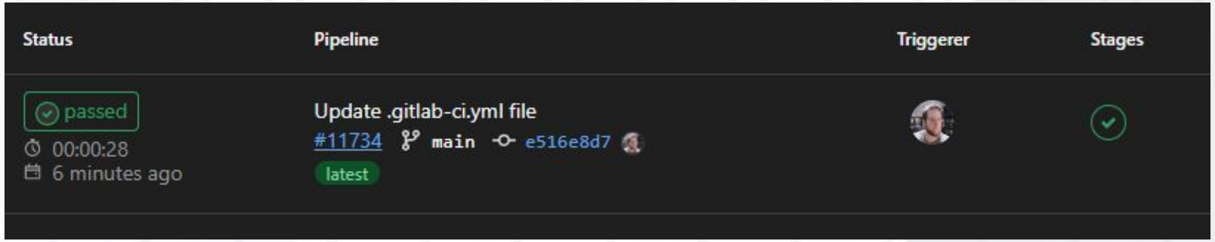
1. Create a pipeline that returns “Hello, Pipeline”
2. Navigate through the pipelines interface to see the results.



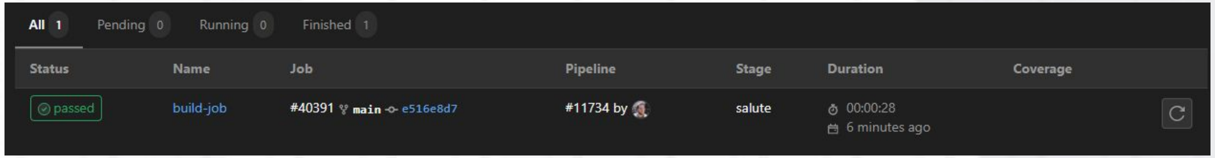
```yaml
stages:           # List of stages for jobs, and their order of execution
  - salute

build-job:        # This job runs in the build stage, which runs first.
  stage: salute
  script:
    - echo "Hello, Pipeline"
```

Pipeline visualization

- Pipeline view
- Pipeline ID
- Branch
- Commit
- Triggerer
- Retry job
- Job logs
- Job view
- Job ID



Status	Pipeline	Triggerer	Stages
passed 00:00:28 6 minutes ago	Update .gitlab-ci.yml file #11734 main e516e8d7 latest		✓



All 1 Pending 0 Running 0 Finished 1

Status	Name	Job	Pipeline	Stage	Duration	Coverage
passed	build-job	#40391 main e516e8d7	#11734 by	salute	00:00:28 6 minutes ago	

GitLab CI keywords

GitLab CI keywords

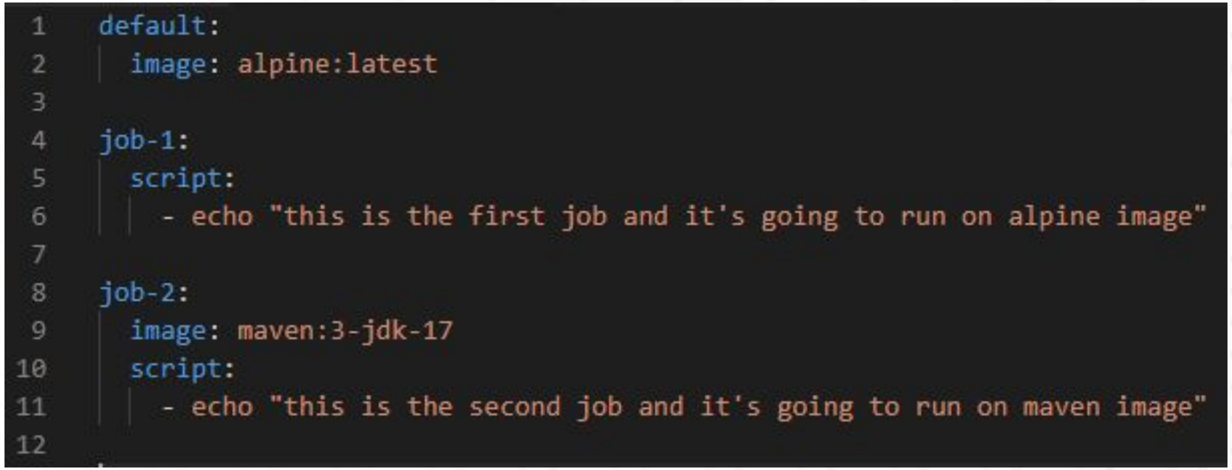
We are going to use keywords to configure and define **how** we want our pipelines and jobs to run.

GitLab CI keywords

Global keywords

default

Set global defaults for other keywords in all the pipeline

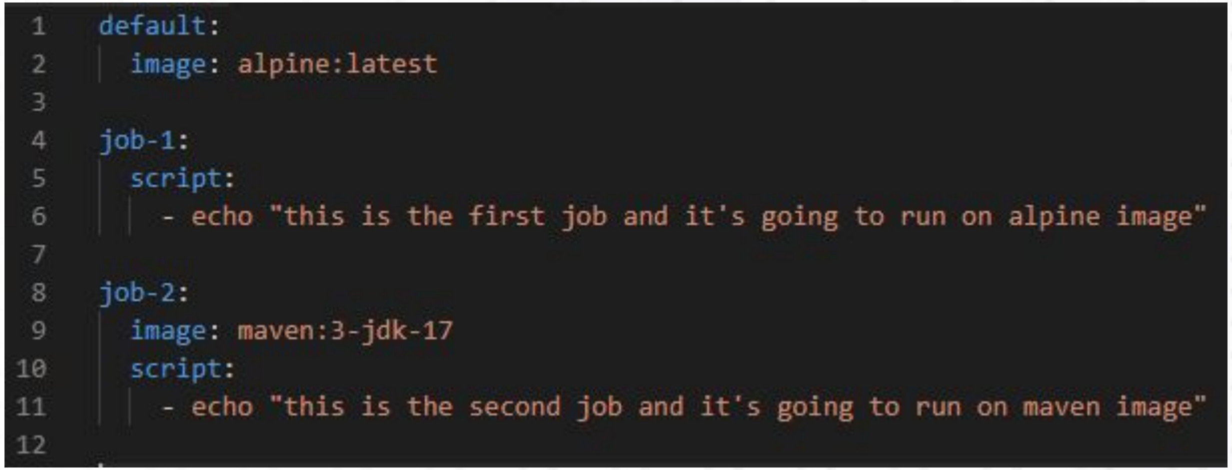


```yaml
default:
  image: alpine:latest

job-1:
  script:
    - echo "this is the first job and it's going to run on alpine image"

job-2:
  image: maven:3-jdk-17
  script:
    - echo "this is the second job and it's going to run on maven image"

```

Global keywords

default

Set global defaults for other keywords in all the pipeline


```yaml
default:
  image: alpine:latest

job-1:
  script:
    - echo "this is the first job and it's going to run on alpine image"

job-2:
  image: maven:3-jdk-17
  script:
    - echo "this is the second job and it's going to run on maven image"

```

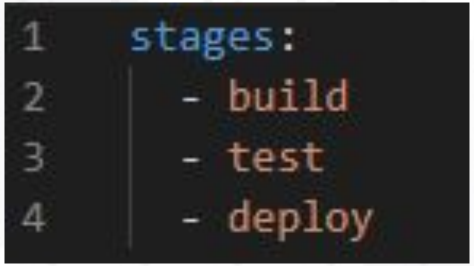
Keywords:

- after_script
- artifacts
- before_script
- cache
- image
- interruptible
- retry
- services
- tags
- timeout

Global keywords

stages

Stages that contain groups of jobs



```yaml
stages:
  - build
  - test
  - deploy
```

Global keywords

stages

- .pre
- build
- test
- deploy
- .post

Job keywords

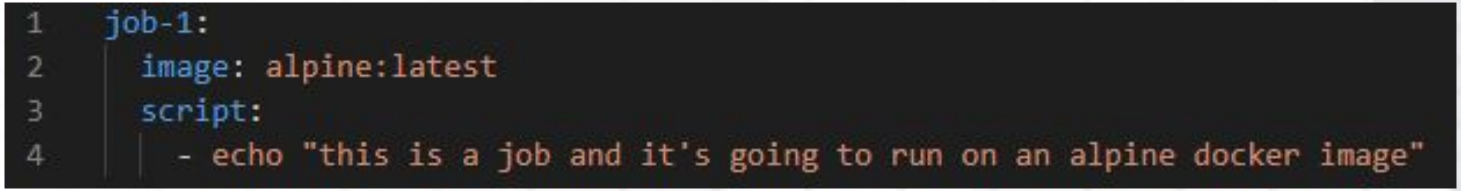
stage

- Which stage a job runs in
- Defaults to *test*
- Can have more than one value

Job keywords

image

Docker image the job runs in.

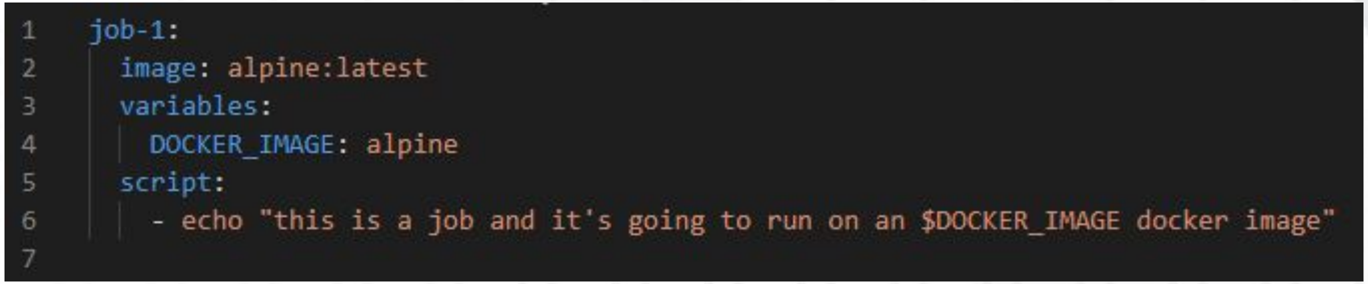


```yaml
job-1:
  image: alpine:latest
  script:
    - echo "this is a job and it's going to run on an alpine docker image"
```

Job keywords

variables

Custom values passed to jobs.

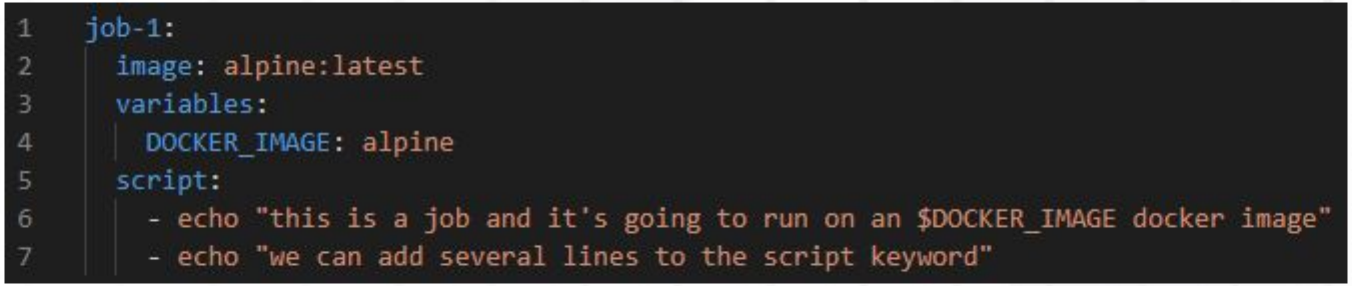


```yaml
job-1:
  image: alpine:latest
  variables:
    DOCKER_IMAGE: alpine
  script:
    - echo "this is a job and it's going to run on an $DOCKER_IMAGE docker image"
```

Job keywords

script

Commands for the runner to execute.



```yaml
job-1:
  image: alpine:latest
  variables:
    DOCKER_IMAGE: alpine
  script:
    - echo "this is a job and it's going to run on an $DOCKER_IMAGE docker image"
    - echo "we can add several lines to the script keyword"
```

Job keywords

when

- Conditions for the job to run
- Boolean values

Job keywords

when

- Conditions for the job to run
- Boolean values
  a. **on_success**

Job keywords

when

- Conditions for the job to run
- Boolean values
  1. **on_success**
  2. **manual**

Job keywords

when

- Conditions for the job to run
- Boolean values
  1. **on_success**
  2. **manual**
  3. **always**

Job keywords

when

- Conditions for the job to run
- Boolean values
    a. **on_success**
    b. **manual**
    c. **always**
    d. **on_failure**

Job keywords

when

- Conditions for the job to run
- Boolean values
    a. **on_success**
    b. **manual**
    c. **always**
    d. **on_failure**
    e. **delayed**

Job keywords

when

- Conditions for the job to run
- Boolean values
    a. **on_success**
    b. **manual**
    c. **always**
    d. **on_failure**
    e. **delayed**
    f. **never**

Exercise 2

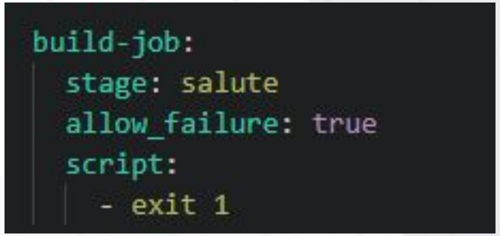
when

- Conditions for the job to run
- Boolean values
    a. **On_success (Que ellos definan algo que nunca falle y qué pasará en consecuencia)**
    ~~b. **manual**~~
    ~~c. **always**~~
    d. **On_failure (Que ellos definan algo que siempre falle y qué pasará en consecuencia)**
    ~~e. **delayed**~~
    f. **never**

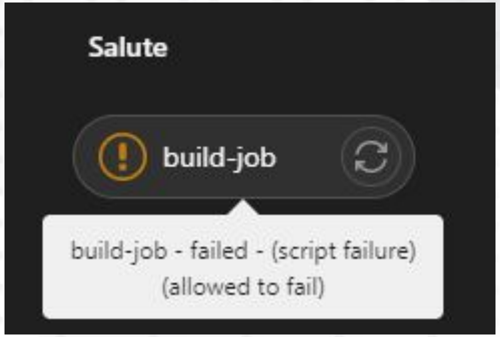
Job keywords

allow_failure

Continue running the pipeline if that job fails



```
build-job:
  stage: salute
  allow_failure: true
  script:
    - exit 1
```



build-job - failed - (script failure)
(allowed to fail)

Job keywords

rules

Select jobs to run based on conditions.

Job keywords

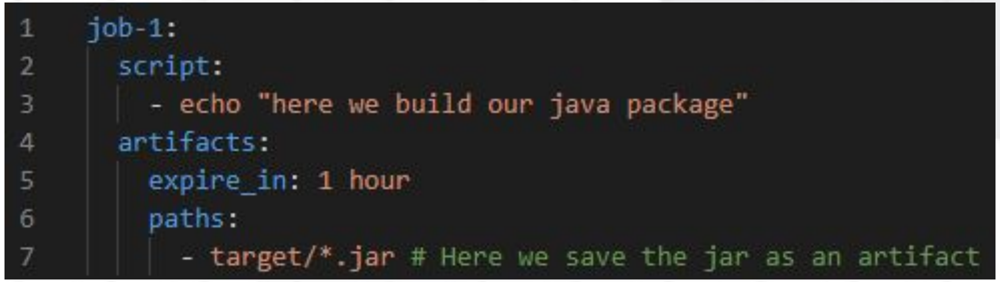
rules

- *if*
- *changes*
- *exists*
- *allow_failure*
- *variables*
- *when*

Job keywords

artifacts

Files and/or directories to save once the job finishes



```yaml
job-1:
  script:
    - echo "here we build our java package"
  artifacts:
    expire_in: 1 hour
    paths:
      - target/*.jar # Here we save the jar as an artifact
```

Exercise: Ci Pipeline

Automate the CI of our application, *greetings*

Exercise: Ci Pipeline

- Clear the steps to build it,
- Have written tests following best practices.

Exercise: Ci Pipeline

- Reduce the time our developers spend with the CI
- Reduce the human errors due to repetitive manual tasks

Exercise: Ci Pipeline

We are going to use our gained knowledge in GitLab CI.

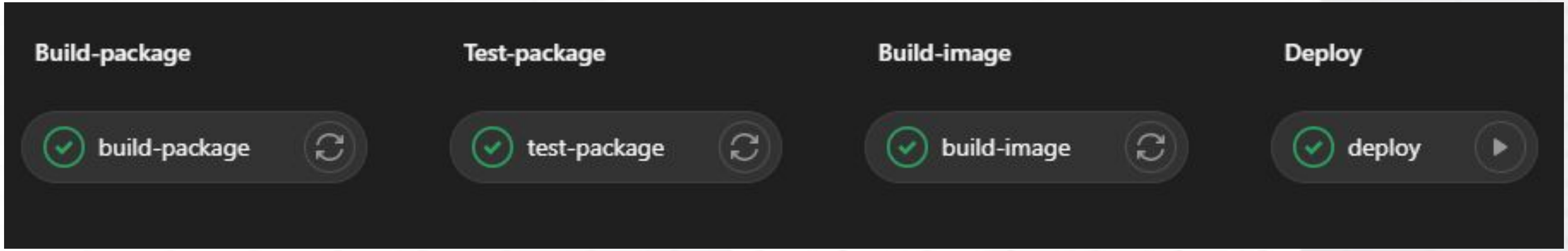
Exercise: Ci Pipeline

The first step will be to define the pipeline:

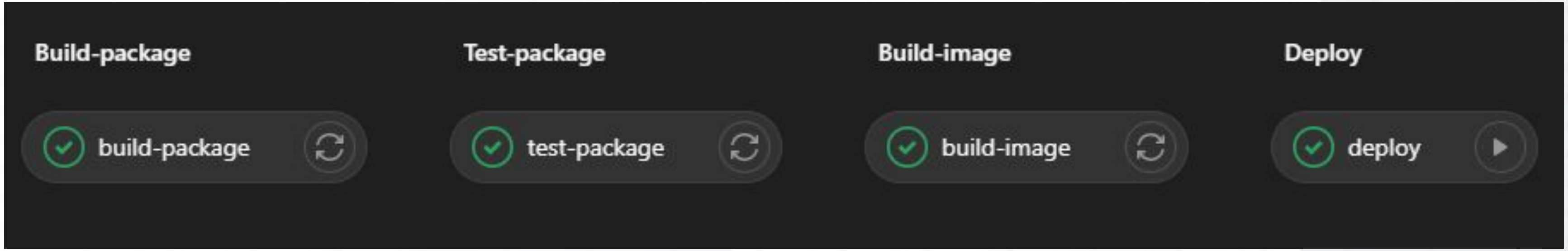
- How many stages?
- How many jobs?
- Which ones?
- What are the dependencies between them?

Exercise: Ci Pipeline

In this case we are going to focus on a basic CI pipeline:



Exercise: Ci Pipeline



- Build job to compile and build our package
- Test job to run our tests
- Build image job to build a docker image based on our compiled code
- Deploy job to deploy to our environment

Exercise: Ci Pipeline. Build package

- What is our programming language?
- What docker image should we use as base?
- What artifacts will we need to use later?
- Do we need to store any kind of variables?

Exercise: Ci Pipeline. Test

When creating this job we need to ask ourselves:

- We will again ask all the questions from the build phase, as they will probably align with this one.
- How do I report my tests to GitLab?
- Do I want to have just a separate test job?
    - If you are clear on how to separate your tests (Unit tests, Integration tests...) and execute them separately then go for it! It will help you diagnose later. You can run them all in the same test stage but with different names.
    - If you are not clear on it, as an initial advice the best option is to go for a generic test job and run all your tests. You can always separate them later!

GitLab CI variables

GitLab CI variables

CI/CD variables are a type of environment variable. We can use them to:

- Control the behavior of jobs and pipelines.
- Store values we want to reuse.
- Avoid hard-coding values in our .gitlab-ci.yml file.

We can use a set of already predefined variables or we can define our own.

GitLab CI variables

A type of environment variable.

GitLab CI variables

- **Control the behavior** of jobs and pipelines.
- **Store values** we want to reuse.
- **Avoid hard-coding** values in our .gitlab-ci.yml file.

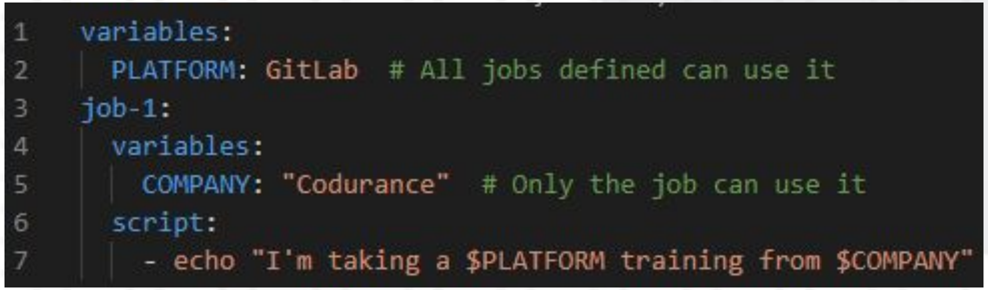
Predefined variables

GitLab CI/CD has a default set of predefined CI/CD variables.

Available in every GitLab CI/CD pipeline.

Custom variables: .gitlab-ci.yaml

- Defined with *variables* keyword.
- Visible in the code and the repository.



```yaml
variables:
  PLATFORM: GitLab  # All jobs defined can use it
job-1:
  variables:
    COMPANY: "Codurance"  # Only the job can use it
  script:
    - echo "I'm taking a $PLATFORM training from $COMPANY"
```

Variable scope

- Job level
    - Only available at the current job
- Pipeline level
    - Available for all jobs in the current pipeline
- Project level
    - Available for all pipelines in the project
    - Possibility to scope by environment
- Group level
    - Available for all pipelines in the projects inside the group
    - Possibility to scope by environment
- Predefined variables
    - Available for all pipelines that run in GitLab

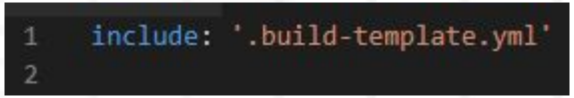
Demo CI pipeline

Advanced elements

Global keywords

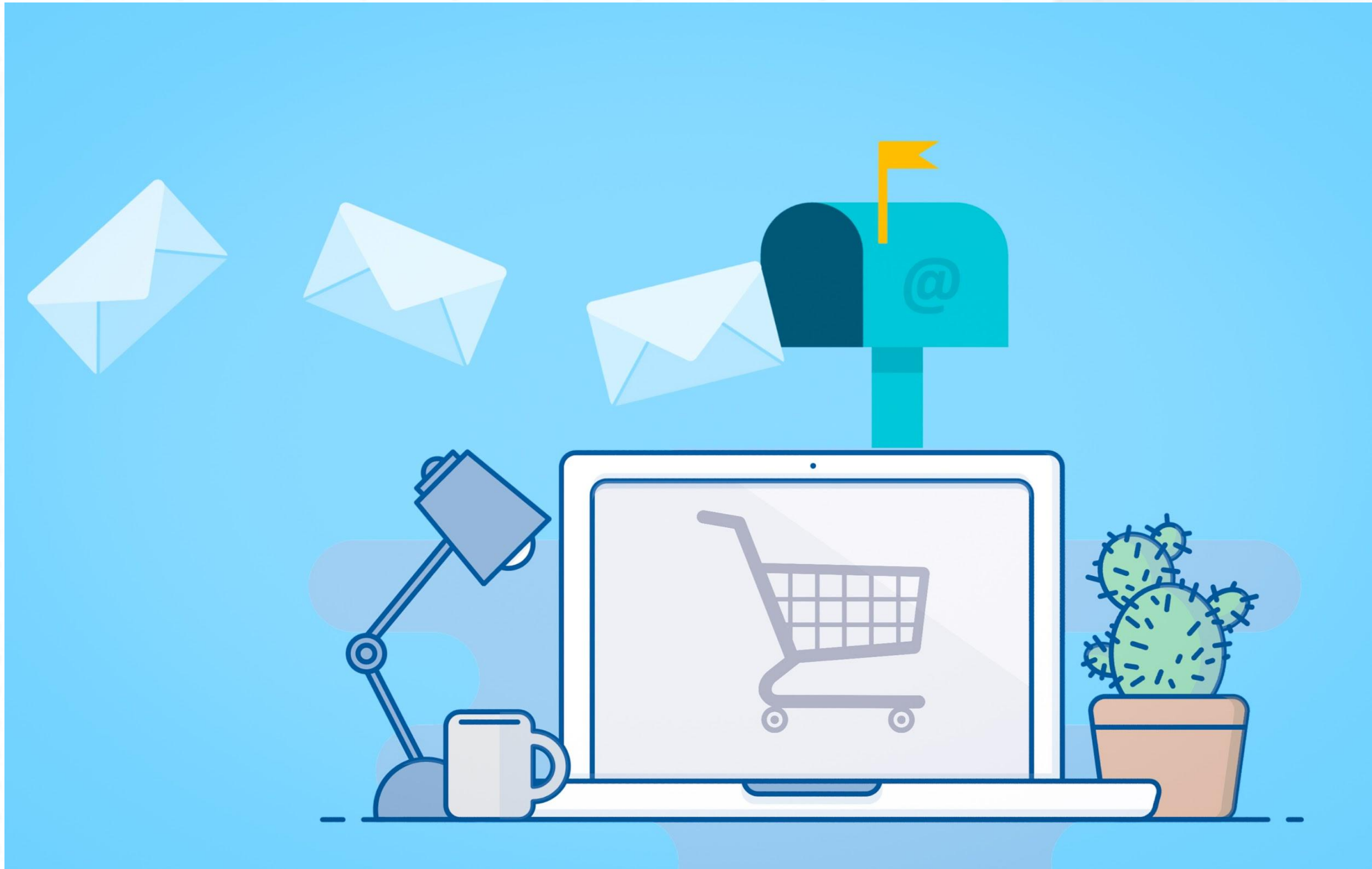
include

Include external yaml files in our CI/CD configuration.

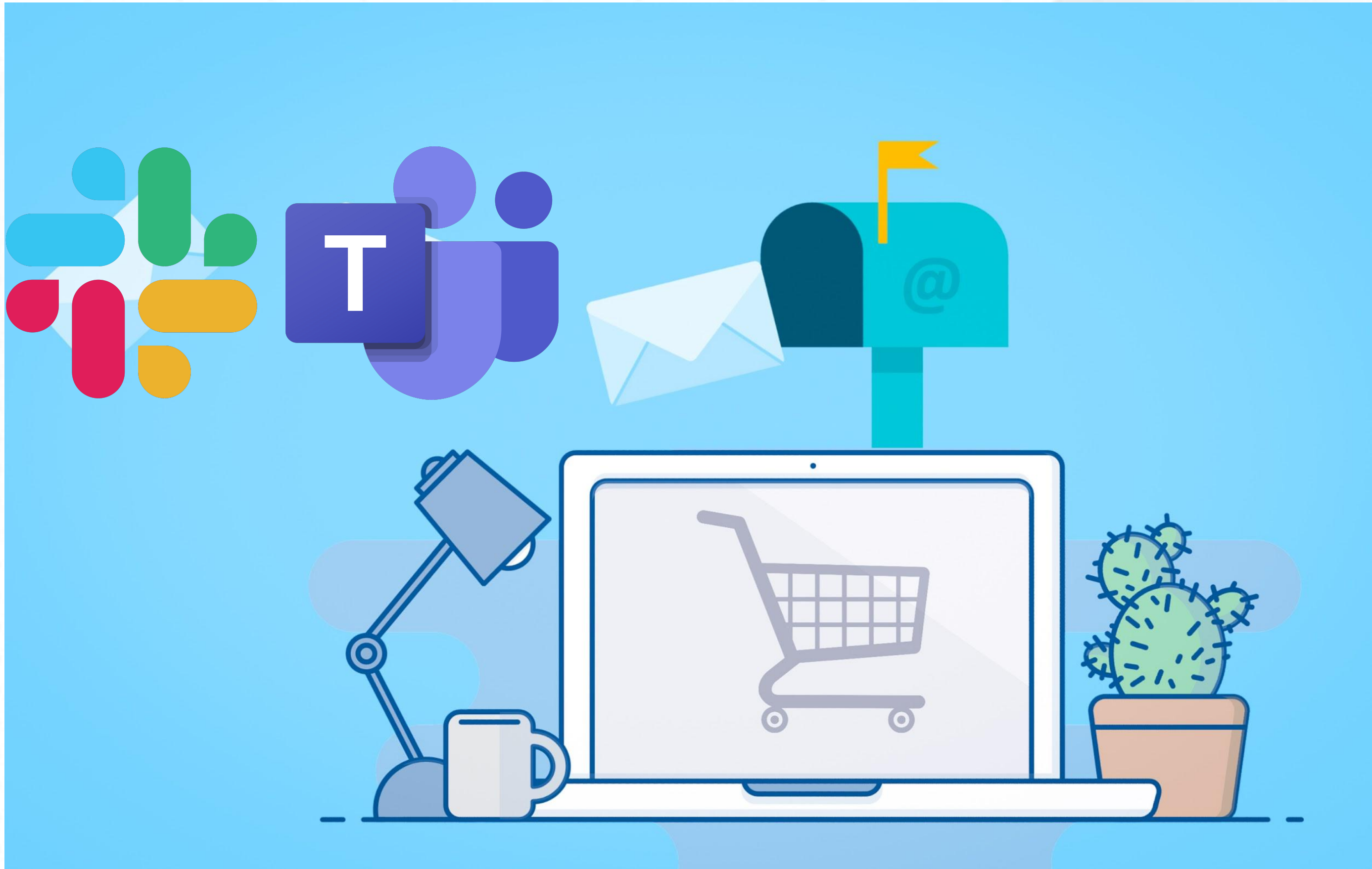


```yaml
include: '.build-template.yml'

```

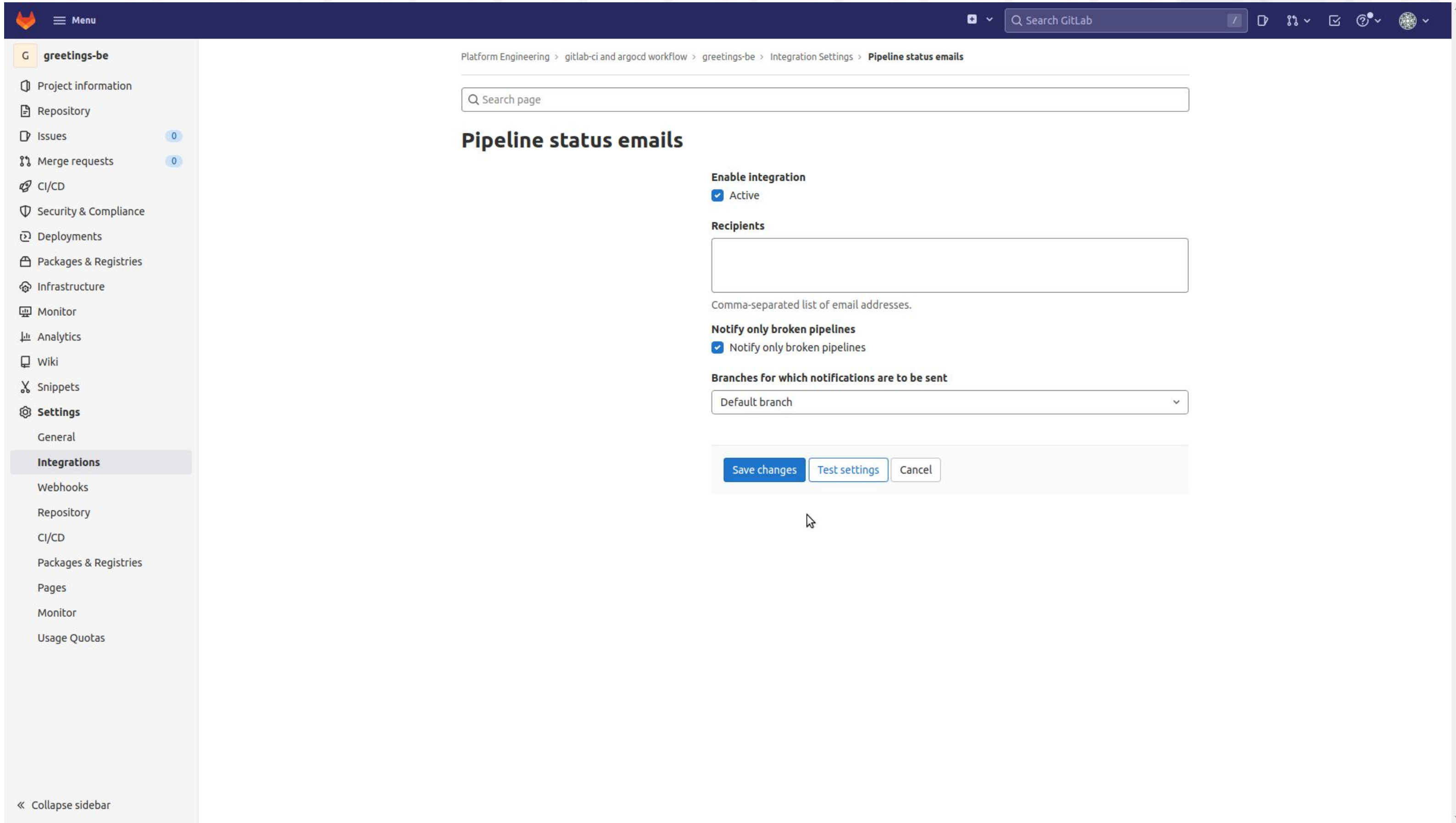
Pipeline notifications



Pipeline notifications



Pipeline notifications



Platform Engineering > gitlab-ci and argocd workflow > greetings-be > Integration Settings > Pipeline status emails

Pipeline status emails

Enable integration

[✓] Active

Recipients

Comma-separated list of email addresses.

Notify only broken pipelines

[✓] Notify only broken pipelines

Branches for which notifications are to be sent

Default branch

Job keywords

environment

The environment a job runs at.

Custom variables: Project/Group settings

We can add CI/CD variables to a group/project’s settings.

Only project members with the Maintainer role can add or update project CI/CD variables. To keep a CI/CD variable secret, we will put it in the project settings, not in the .gitlab-ci.yml file.

We can add simple variables or full files as variables, such as SSH keys or parameters files.

Custom variables: Project/Group settings

To add or update variables in the project settings:

- Go to your project’s Settings > CI/CD > Variables
- Select the Add Variable button and fill in the details:
  - Key: the key for our variable
  - Value: the content of our variable
  - Type: File or Variable.
  - Environment scope: Optional. All, or specific environments.
  - Protect variable Optional. If selected, the variable is only available in pipelines that run on protected branches or protected tags.
  - Mask variable Optional. If selected, the variable’s Value is masked in job logs. The variable fails to save if the value does not meet the masking requirements.

Variable precedence

1. These all have the same (highest) precedence:
    - Trigger variables.
    - Scheduled pipeline variables.
    - Manual pipeline run variables.
    - Variables added when creating a pipeline with the API.
2. Project variables.
3. Group variables.
4. Instance variables.
5. Inherited variables.
6. Variables defined in jobs in the .gitlab-ci.yml file.
7. Variables defined outside of jobs (globally) in the .gitlab-ci.yml file.
8. Deployment variables.
9. Predefined variables.

Pipeline images in our own GitLab Container Registry

M

Job keywords

needs

This keyword will allow us to execute jobs out of order, ignoring stage ordering and allowing some jobs to run without waiting all the jobs from a previous stage to complete.

If we leave this keyword empty it will mean that the job is set to start as soon as the pipeline is created.

Job keywords

dependencies

We will use the dependencies keyword to define a list of jobs to fetch artifacts from. We can also set a job to download no artifacts at all.

If we do not use dependencies, all artifacts from previous stages are passed to each job.

If we leave the keyword empty we will be configuring the job to not download any artifacts.

Job keywords

before_script / after_script

We will use these keywords to define an array of commands that should run before/after each job's *script* commands.

- *before*: they are concatenated with the *script* commands and they run in a single shell
- *after*: they run even if the job fails. These scripts execute in a new shell, separate from the *before_script* and *script* commands. That means they don't have access to the changes done in those commands.

Trigger keyword

M

Merge requests and working with features

M

Code coverage

M

Job keywords

coverage

This keyword will be used

Badges

M

Security configuration

M

GitLab CI registries

GitLab Container Registry

The GitLab Container Registry is a secure and private registry for container images. It’s built on open source software and completely integrated within GitLab.

The stored Images follow this naming convention:
<registry URL>/<namespace>/<project>/<image>

We will use GitLab CI/CD to create and publish images from the pipelines. In order to do that we will have to authenticate with the Container Registry. That’s when the predefined variables enter in place with these two options:

- docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
- docker login -u $CI_REGISTRY_USER -p $CI_JOB_TOKEN $CI_REGISTRY

Pull images from GitLab Container Registry

1. Create access token

We will access into out GitLab profile and we will create an access token with permissions to read and write into the registry:

2. Log into the docker registry

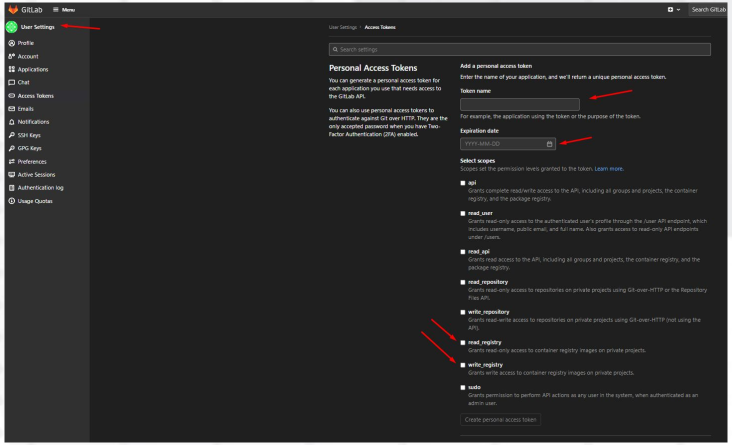
We will login into the docker registry with our GitLab username and our newly created token:

```
amorell@LAPTOP-BIT56HGB:~$ docker login -u alex [MYREGISTRY]
Password:
Login Succeeded
```

3. Pull docker image

We will pull the docker image with the standard commands:

```
docker pull [MYREGISTRY]/[MYIMAGE]:[MYTAG]
```



Personal Access Tokens

You can generate a personal access token for each application you use that needs access to the GitLab API.

You can also use personal access tokens to authenticate against Git over HTTP. They are the only accepted password when you have Two-Factor Authentication (2FA) enabled.

Add a personal access token

Enter the name of your application, and we'll return a unique personal access token.

Token name

For example, the application using the token or the purpose of the token.

Expiration date

YYYY-MM-DD

Select scopes

Scopes set the permission levels granted to the token. Learn more.

- **api** Grants complete read/write access to the API, including all groups and projects, the container registry, and the package registry.
- **read_user** Grants read-only access to the authenticated user's profile through the /user API endpoint, which includes username, public email, and full name. Also grants access to read-only API endpoints under /users.
- **read_api** Grants read access to the API, including all groups and projects, the container registry, and the package registry.
- **read_repository** Grants read-only access to repositories on private projects using Git-over-HTTP or the Repository Files API.
- **write_repository** Grants read-write access to repositories on private projects using Git-over-HTTP (not using the API).
- **read_registry** Grants read-only access to container registry images on private projects.
- **write_registry** Grants write access to container registry images on private projects.
- **sudo** Grants permission to perform API actions as any user in the system, when authenticated as an admin user.

GitLab Package Registry

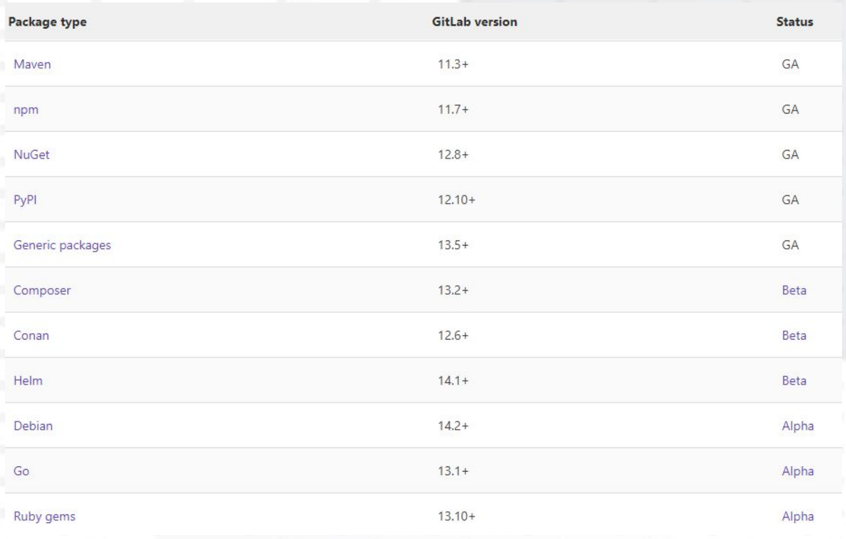
With the GitLab Package Registry, we can use GitLab as a private or public registry for a variety of supported package managers. We can also publish and share packages, which can be consumed as a dependency in downstream projects.

In order to view packages for your project or group:

1. Go to the project or group.
2. Go to Packages & Registries > Package Registry.

We will be able to search, sort, and filter packages on this page, and also share your search results by copying and pasting the URL from our browser.

The package manager supports several formats:



Package type	GitLab version	Status
Maven	11.3+	GA
npm	11.7+	GA
NuGet	12.8+	GA
PyPI	12.10+	GA
Generic packages	13.5+	GA
Composer	13.2+	Beta
Conan	12.6+	Beta
Helm	14.1+	Beta
Debian	14.2+	Alpha
Go	13.1+	Alpha
Ruby gems	13.10+	Alpha

Exercise: Advanced Ci Pipeline. Build image

When creating this job we need to ask ourselves:

- Do I need any Docker specific variables?
- When do I want to built the image, on main branch after merge requests or also on features?
- Do I have a Dockerfile already, and where is it stored?
- Where do I want to store my built Image? Will it be my own registry or GitLab registry?

GitLab CI Training

Resources

GitLab CI Documentation

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

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

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

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

codurance.com

Thank you

Fran Iglesias
Software Craftperson

fran.iglesias@codurance.com
in Fran Iglesias

Mauro Chojrin
Software Craftperson

mauro.chojrin@codurance.com
in Mauro Chrojrin