GitLab CI: Run jobs sequentially, in parallel or build a custom pipeline

Ivan Nemytchenko (https://twitter.com/inemation) Jul 29, 2016

Let's assume that you don't know anything about what Continuous Integration is and why it's needed. Or, you just forgot. Anyway, we're starting from scratch here.

Imagine that you work on a project, where all the code consists of two text files. Moreover, it is super-critical that the concatenation of these two files contains the phrase "Hello world."

If there's no such phrase, the whole development team stays without a salary for a month. Yeah, it is that serious!

The most responsible developer wrote a small script to run every time we are about to send our code to customers. The code is pretty sophisticated:

```
cat file1.txt file2.txt | grep -q "Hello world"
```

The problem is that there are ten developers in the team, and, you know, human factors can hit hard.

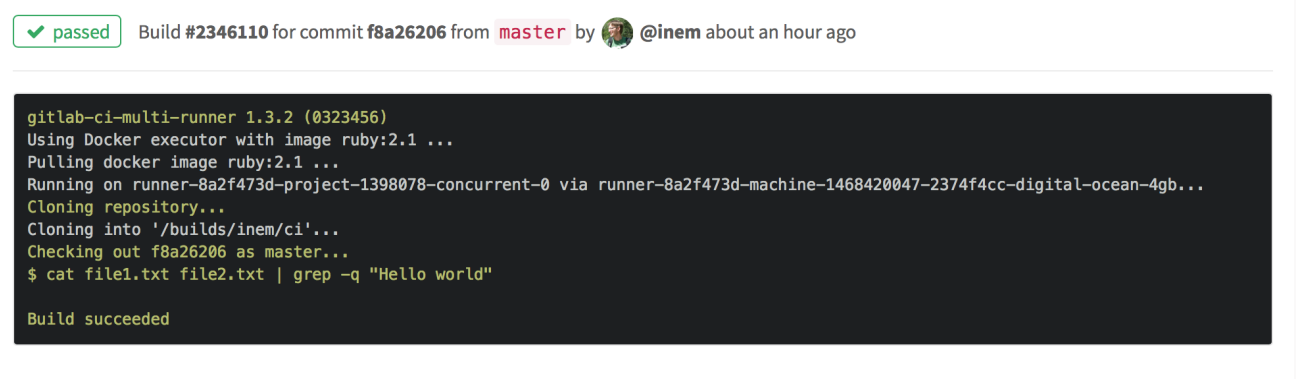
A week ago, a new guy forgot to run the script and three clients got broken builds. So you decided to solve the problem once and for all. Luckily, your code is already on GitLab, and you remember that there is a built-in CI system (https://about.gitlab.com/gitlab-ci/). Moreover, you heard at a conference that people use CI to run tests…

Run our first test inside CI

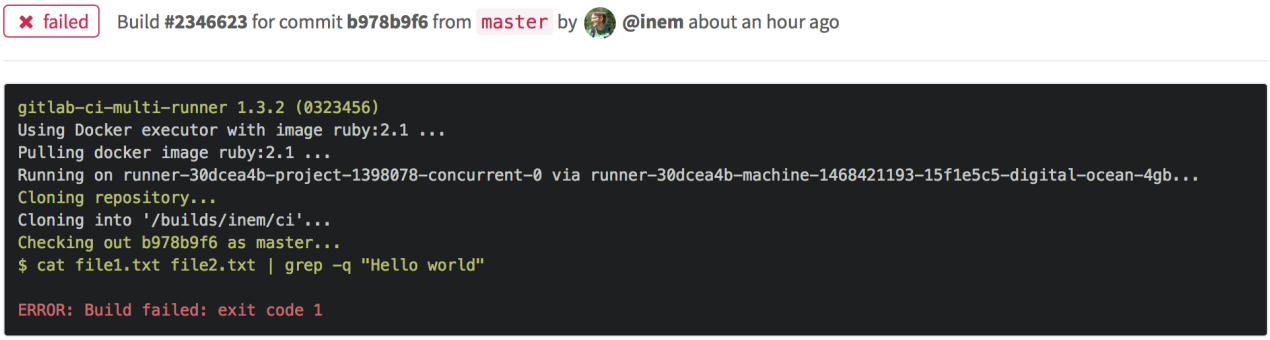
After a couple minutes to find and read the docs, it seems like all we need is these two lines of code in a file called `.gitlab-ci.yml`:

```
test:
  script: cat file1.txt file2.txt | grep -q 'Hello world'
```

Committing it, and hooray! Our build is successful:



Let's change "world" to "Africa" in the second file and check what happens:



The build fails as expected!

Okay, we now have automated tests here! GitLab CI will run our test script every time we push new code to the repository.

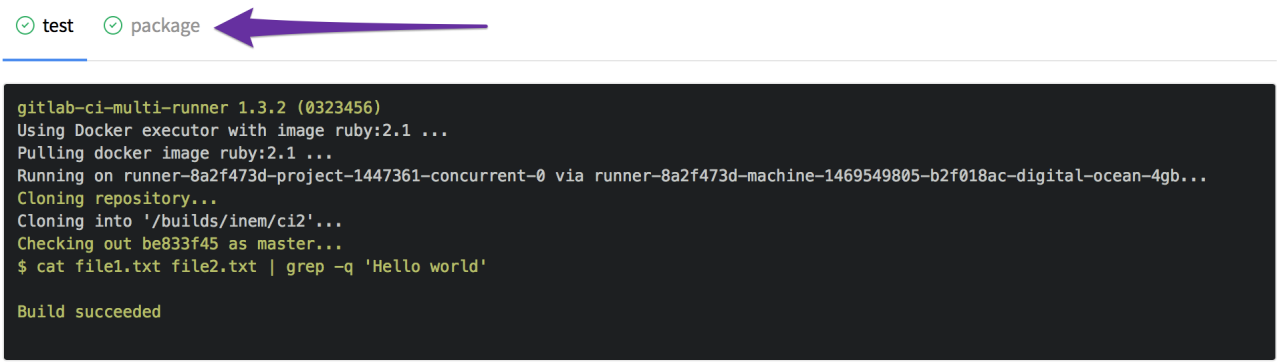
Make results of builds downloadable

The next business requirement is to package the code before sending it to our customers. Let's automate that as well!

All we need to do is define another job for CI. Let's name the job "package":

```
test:
  script: cat file1.txt file2.txt | grep -q 'Hello world'

package:
  script: cat file1.txt file2.txt | gzip > package.gz
```

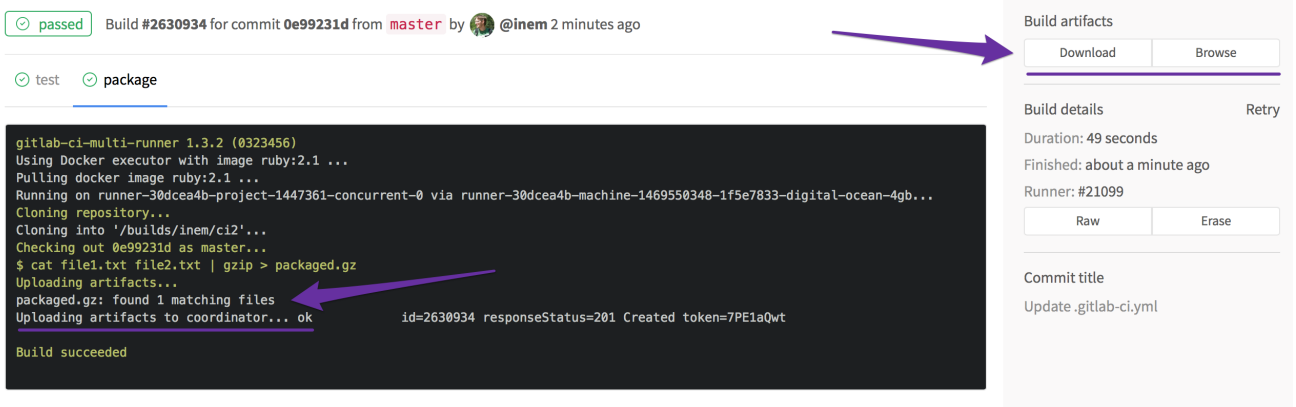
We have two tabs now:



However, we forgot to specify that the new file is a build *artifact*, so that it could be downloaded. We can fix it by adding an `artifacts` section:

```
test:
  script: cat file1.txt file2.txt | grep -q 'Hello world'

package:
  script: cat file1.txt file2.txt | gzip > packaged.gz
  artifacts:
    paths:
    - packaged.gz
```

Checking… It is there:



Perfect! However, we have a problem to fix: the jobs are running in parallel, but we do not want to package our application if our tests fail.

Run jobs sequentially

We only want to run the 'package' job if the tests are successful. Let's define the order by specifying `stages`:

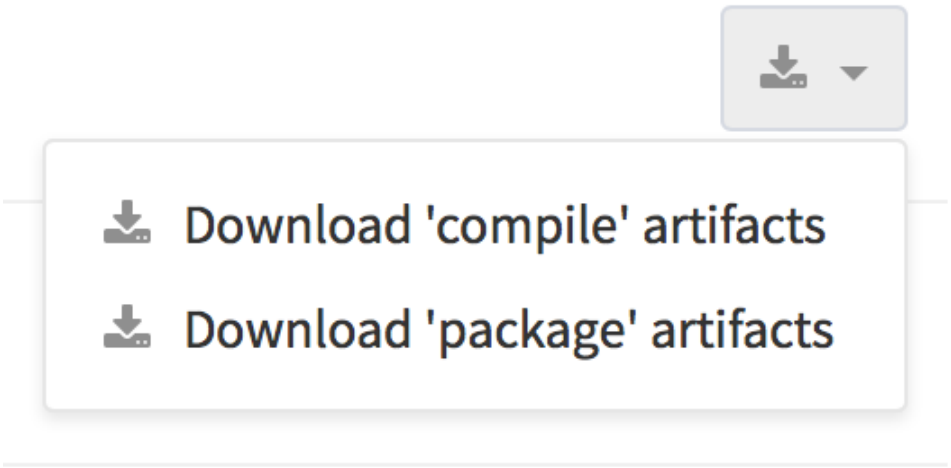
```
stages:
  - test
  - package

test:
  stage: test
  script: cat file1.txt file2.txt | grep -q 'Hello world'

package:
  stage: package
  script: cat file1.txt file2.txt | gzip > packaged.gz
  artifacts:
    paths:
    - packaged.gz
```

That should be good!

Also, we forgot to mention, that compilation (which is represented by concatenation in our case) takes a while, so we don't want to run it twice. Let's define a separate step for it:

```
stages:
  - compile
  - test
  - package

compile:
  stage: compile
  script: cat file1.txt file2.txt > compiled.txt
  artifacts:
    paths:
    - compiled.txt

test:
  stage: test
  script: cat compiled.txt | grep -q 'Hello world'

package:
  stage: package
  script: cat compiled.txt | gzip > packaged.gz
  artifacts:
    paths:
    - packaged.gz
```

Let's take a look at our artifacts:



Hmm, we do not need that "compile" file to be downloadable. Let's make our temporary artifacts expire by setting `expire_in` to '20 minutes':

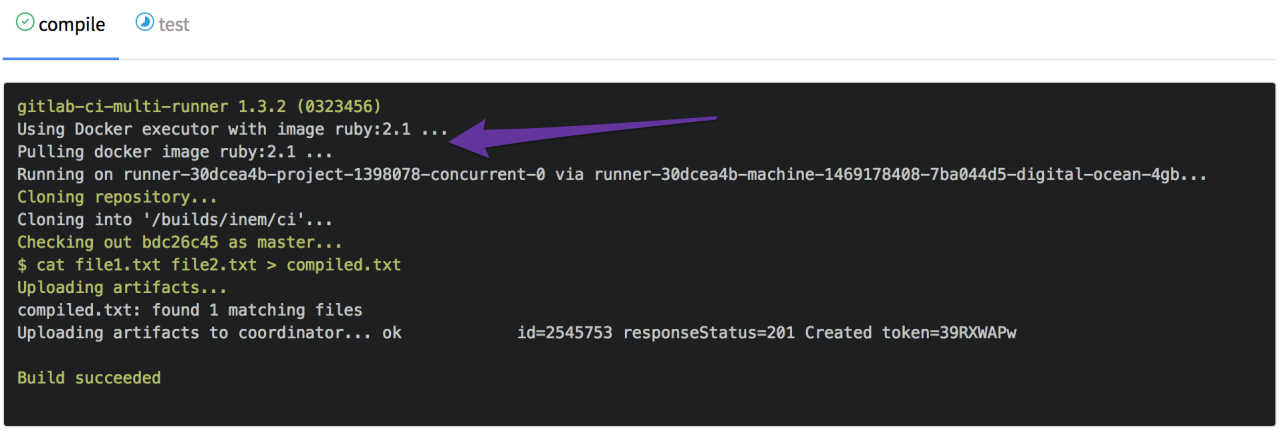
```
compile:
  stage: compile
  script: cat file1.txt file2.txt > compiled.txt
  artifacts:
    paths:
    - compiled.txt
    expire_in: 20 minutes
```

Now our config looks pretty impressive:

- We have three sequential stages to compile, test, and package our application.
- We are passing the compiled app to the next stages so that there's no need to run compilation twice (so it will run faster).
- We are storing a packaged version of our app in build artifacts for further usage.

Learning which Docker image to use

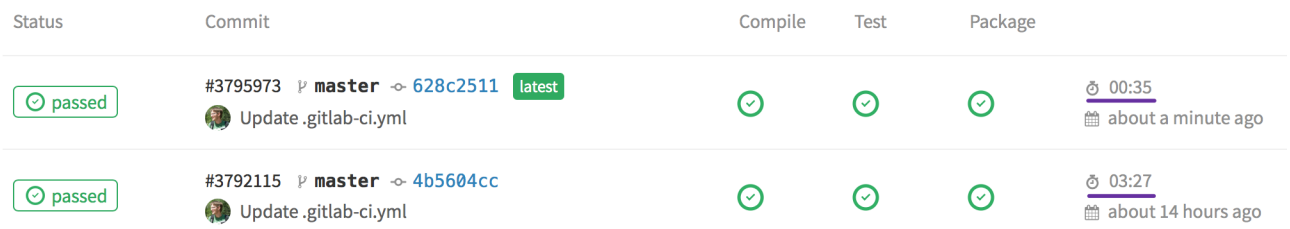
So far so good. However, it appears our builds are still slow. Let's take a look at the logs.



Wait, what is this? Ruby 2.1?

Why do we need Ruby at all? Oh, GitLab.com uses Docker images to run our builds (/2016/04/05/shared-runners/), and by default (/gitlab-com/settings/#shared-runners) it uses the `ruby:2.1` (https://hub.docker.com/_/ruby/) image. For sure, this image contains many packages we don't need. After a minute of googling, we figure out that there's an image called `alpine` (https://hub.docker.com/_/alpine/) which is an almost blank Linux image.

OK, let's explicitly specify that we want to use this image by adding `image: alpine` to `.gitlab-ci.yml`. Now we're talking! We shaved almost 3 minutes off:



It looks like there's (https://hub.docker.com/_/mysql/) a lot of (https://hub.docker.com/_/python/) public images (https://hub.docker.com/_/java/) around (https://hub.docker.com/_/php/). So we can just grab one for our technology stack. It makes sense to specify an image which contains no extra software because it minimizes download time.

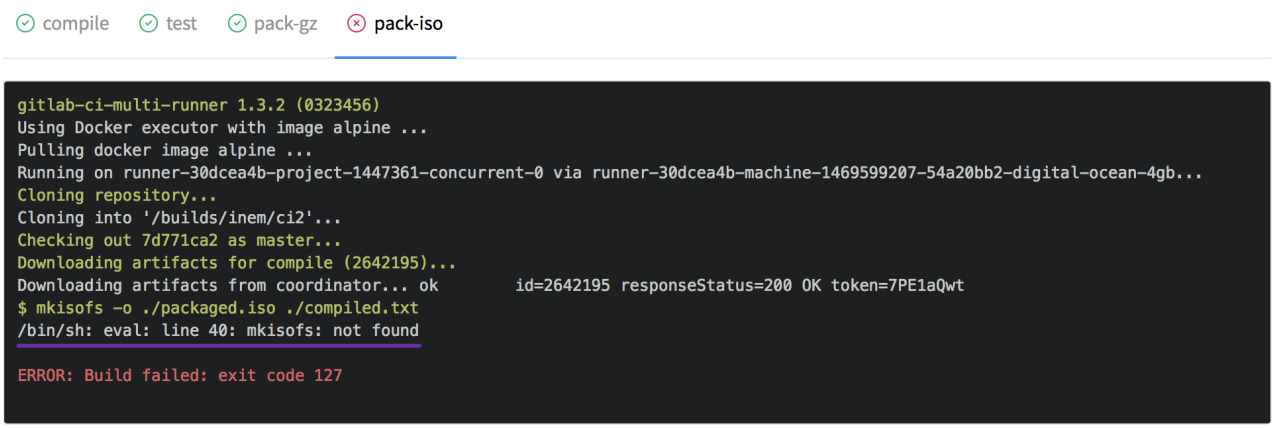
Dealing with complex scenarios

So far so good. However, let's suppose we have a new client who wants us to package our app into `.iso` image instead of `.gz` Since CI does the whole work, we can just add one more job to it. ISO images can be created using the mkisofs (http://linuxcommand.org/man_pages/mkisofs8.html) command. Here's how our config should look:

```
image: alpine

stages:
  - compile
  - test
  - package

# ... "compile" and "test" jobs are skipped here for the sake of compactness

pack-gz:
  stage: package
  script: cat compiled.txt | gzip > packaged.gz
  artifacts:
    paths:
    - packaged.gz

pack-iso:
  stage: package
  script:
  - mkisofs -o ./packaged.iso ./compiled.txt
  artifacts:
    paths:
    - packaged.iso
```

Note that job names shouldn't necessarily be the same. In fact if they were the same, it wouldn't be possible to make the jobs run in parallel inside the same stage. Hence, think of same names of jobs & stages as coincidence.

Anyhow, the build is failing:



The problem is that `mkisofs` is not included in the `alpine` image, so we need to install it first.

Dealing with missing software/packages

According to the Alpine Linux website (https://pkgs.alpinelinux.org/contents?file=mkisofs&path=&name=&branch=&repo=&arch=x86) `mkisofs` is a part of the `xorriso` and `cdrkit` packages. These are the magic commands that we need to run to install a package:

```
echo "ipv6" >> /etc/modules  # enable networking
apk update                   # update packages list
apk add xorriso              # install package
```

For CI, these are just like any other commands. The full list of commands we need to pass to `script` section should look like this:

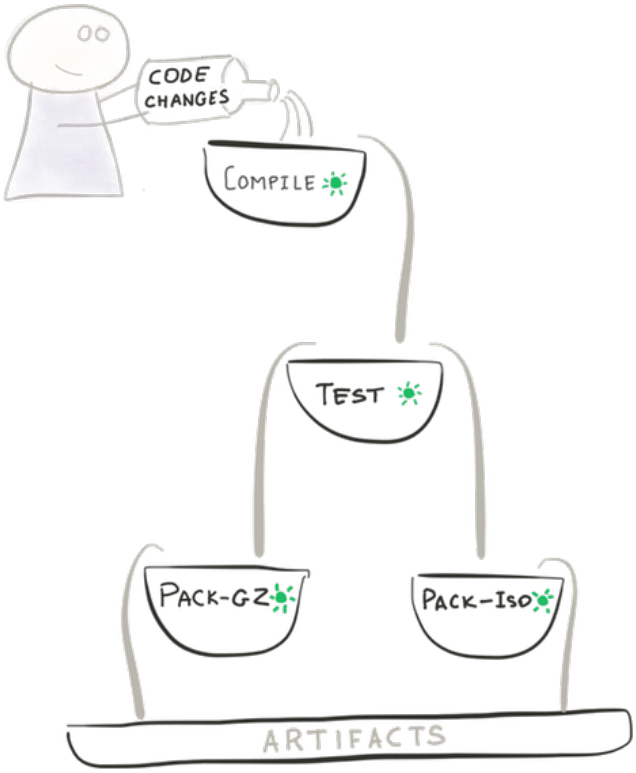
```
script:
- echo "ipv6" >> /etc/modules
- apk update
- apk add xorriso
- mkisofs -o ./packaged.iso ./compiled.txt
```

However, to make it semantically correct, let's put commands related to package installation in `before_script`. Note that if you use `before_script` at the top level of a configuration, then the commands will run before all jobs. In our case, we just want it to run before one specific job.

Our final version of `.gitlab-ci.yml`:

```
image: alpine

stages:
  - compile
  - test
  - package

compile:
  stage: compile
  script: cat file1.txt file2.txt > compiled.txt
  artifacts:
    paths:
    - compiled.txt
    expire_in: 20 minutes

test:
  stage: test
  script: cat compiled.txt | grep -q 'Hello world'

pack-gz:
  stage: package
  script: cat compiled.txt | gzip > packaged.gz
  artifacts:
    paths:
    - packaged.gz

pack-iso:
  stage: package
  before_script:
  - echo "ipv6" >> /etc/modules
  - apk update
  - apk add xorriso
  script:
  - mkisofs -o ./packaged.iso ./compiled.txt
  artifacts:
    paths:
    - packaged.iso
```

Wow, it looks like we have just created a pipeline! We have three sequential stages, but jobs `pack-gz` and `pack-iso`, inside the `package` stage, are running in parallel:



Summary

There's much more to cover but let's stop here for now. I hope you liked this short story. All examples were made intentionally trivial so that you could learn the concepts of GitLab CI without being distracted by an unfamiliar technology stack. Let's wrap up what we have learned:

1. To delegate some work to GitLab CI you should define one or more jobs (http://docs.gitlab.com/ce/ci/yaml/README.html#jobs) in `.gitlab-ci.yml`.
2. Jobs should have names and it's your responsibility to come up with good ones.
3. Every job contains a set of rules & instructions for GitLab CI, defined by special keywords.
4. Jobs can run sequentially, in parallel, or you can define a custom pipeline.
5. You can pass files between jobs and store them in build artifacts so that they can be downloaded from the interface.

Below is the last section containing a more formal description of terms and keywords we used, as well as links to the detailed description of GitLab CI functionality.

Keywords description & links to the documentation

Keyword/term	Description
.gitlab-ci.yml (http://docs.gitlab.com/ce/ci/yaml/README.html#gitlab-ci-yml)	File containing all definitions of how your project should be built
script (http://docs.gitlab.com/ce/ci/yaml/README.html#script)	Defines a shell script to be executed
before_script (http://docs.gitlab.com/ce/ci/yaml/README.html#before_script)	Used to define the command that should be run before (all) jobs

Keyword/term	Description
image (http://docs.gitlab.com/ce/ci/docker/using_docker_images.html#what-is-image)	Defines what docker image to use
stage (http://docs.gitlab.com/ce/ci/yaml/README.html#stages)	Defines a pipeline stage (default: `test`)
artifacts (http://docs.gitlab.com/ce/ci/yaml/README.html#artifacts)	Defines a list of build artifacts
artifacts:expire_in (http://docs.gitlab.com/ce/ci/yaml/README.html#artifactsexpire_in)	Used to delete uploaded artifacts after the specified time
pipelines (http://docs.gitlab.com/ee/ci/pipelines.html#pipelines)	A pipeline is a group of builds that get executed in stages (batches)

Don't miss these GitLab CI stories as well:

- Migrating from Jenkins to GitLab CI (https://about.gitlab.com/2016/07/22/building-our-web-app-on-gitlab-ci/)
- Decreasing build time from 8 minutes 33 seconds to just 10 seconds (http://beenje.github.io/blog/posts/gitlab-ci-and-conda/)