Gitlab-CI: Setting up Continuous Integration for a Gitlab Project Including Creating a Runner

Duncan C. White, d.white@imperial.ac.uk

Dept of Computing, Imperial College London

1st Feb 2016

Duncan White (Imperial)

Gitlab-CI: Setting up Continuous Integration

1st Feb 2016 1 / 15

A (1) > A (2) > A

 Gitlab-Cl is a new Gitlab feature, enabling you to setup automatic actions (eg testing, integration with other components, package building etc) on your Gitlab projects, that run every time you push changes up.

- Gitlab-Cl is a new Gitlab feature, enabling you to setup automatic actions (eg testing, integration with other components, package building etc) on your Gitlab projects, that run every time you push changes up.
- You can add CI to an existing Gitlab project, but this tutorial creates a new Gitlab project, containing a simplified version of a C-based program that flattens nested mailing lists.

- Gitlab-Cl is a new Gitlab feature, enabling you to setup automatic actions (eg testing, integration with other components, package building etc) on your Gitlab projects, that run every time you push changes up.
- You can add CI to an existing Gitlab project, but this tutorial creates a new Gitlab project, containing a simplified version of a C-based program that flattens nested mailing lists.
- Log in to the gitlab.doc.ic.ac.uk web interface and create a new project called **mini-list-flattening**.

- Gitlab-Cl is a new Gitlab feature, enabling you to setup automatic actions (eg testing, integration with other components, package building etc) on your Gitlab projects, that run every time you push changes up.
- You can add CI to an existing Gitlab project, but this tutorial creates a new Gitlab project, containing a simplified version of a C-based program that flattens nested mailing lists.
- Log in to the gitlab.doc.ic.ac.uk web interface and create a new project called **mini-list-flattening**.
- Then populate it as follows:

```
pushd /tmp
wget http://www.doc.ic.ac.uk/~dcw/mini-list-flattening.tgz
tar xzf mini-list-flattening.tgz
cd mini-list-flattening
```

• Then follow the "Existing folder" instructions on the newly created Gitlab project page. For me, these were:

```
git init
git remote add origin git@gitlab.doc.ic.ac.uk:dcw/mini-list-flattening.git
git add .
git commit -m "first commit"
git push -u origin master
```

Duncan White (Imperial) Gitlab-CI: Setting up Continuous Integration

```
1st Feb 2016 2 / 15
```

 In Gitlab, click again on the Project button and you should see the results of the first commit. On Project Settings, check that the Builds feature is enabled, and, further down, that your project has a CI Token. You'll need to copy this token later.

- In Gitlab, click again on the **Project** button and you should see the results of the first commit. On **Project Settings**, check that the **Builds** feature is enabled, and, further down, that your project has a **CI Token**. You'll need to copy this token later.
- In your new **mini-list-flattening** repo directory, look around and see what the code does. In particular, read the README and do what it says to compile and run the test program.

・ロン ・聞と ・ ほと ・ ほと

- In Gitlab, click again on the **Project** button and you should see the results of the first commit. On **Project Settings**, check that the **Builds** feature is enabled, and, further down, that your project has a **CI Token**. You'll need to copy this token later.
- In your new **mini-list-flattening** repo directory, look around and see what the code does. In particular, read the README and do what it says to compile and run the test program.
- No Gitlab project will use CI unless you set up a YAML file called **.gitlab-ci.yml** defining the actions to run. You will spend a lot of time editing/committing/pushing this file, until it works.

イロト イポト イヨト イヨト 二日

- In Gitlab, click again on the **Project** button and you should see the results of the first commit. On **Project Settings**, check that the **Builds** feature is enabled, and, further down, that your project has a **CI Token**. You'll need to copy this token later.
- In your new mini-list-flattening repo directory, look around and see what the code does. In particular, read the README and do what it says to compile and run the test program.
- No Gitlab project will use CI unless you set up a YAML file called **.gitlab-ci.yml** defining the actions to run. You will spend a lot of time editing/committing/pushing this file, until it works.
- After a lot of failures, the first roughly correct version read:

```
before_script:
  - sudo aptrget update -qq && sudo apt-get install -y -qq gcc make
  - which gcc
runtests:
  script:
  # compile it up
  - export TOOLDIR=$HOME/c-tools
  - export ARCH=x86_64
  - make
  # and run the tests..
```

```
- make test
```

イロト 不得 トイヨト イヨト 二日

 In the YML file, there can be any number of sections. The before_script section is special, and means do each command in the list at the beginning of every test build.

A = A = A = A = A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A
 A

- In the YML file, there can be any number of sections. The **before_script** section is special, and means *do each command in the list at the beginning of every test build*.
- The second section can be named whatever you like, here **runtests** was my choice. The **script** tag means *run a sequence of commands*: first we compile the software, then we test it:

```
runtests:
```

script:

- # compile it up
- export TOOLDIR=\$HOME/c-tools

```
- export ARCH=x86_64
```

- make
- # and run the tests..

```
- make test
```

- The commands are exactly those (bash syntax) commands that the README file told us to use to build the program and run the tests. They assume that a \$HOME/c-tools directory exists, containing a few useful modules that I use in most C projects.
- Create yourself a .gitlab-ci.yml file with the full contents from page 3, then git add it, git commit it and git push it up.

- When you push this file up to your remote repo on Gitlab, Gitlab will automatically enable CI facilities on the project.
- In the Gitlab UI, there's a **Builds** menu item, click on it and you will see that it attempted to run a build, initially this will be marked as Pending.
- Click on the Pending Build and you'll see that it's Pending because you haven't yet created and registered a **Runner**.
- A **Runner** is a special test machine, belonging to you and running special software, that Gitlab-CI will use to run your actions. We'll see how to set up and customize the Runner later in this tutorial. In particular, we'll need to ensure that the \$HOME/c-tools directory exists with the right contents.

- When you push this file up to your remote repo on Gitlab, Gitlab will automatically enable CI facilities on the project.
- In the Gitlab UI, there's a **Builds** menu item, click on it and you will see that it attempted to run a build, initially this will be marked as Pending.
- Click on the Pending Build and you'll see that it's Pending because you haven't yet created and registered a **Runner**.
- A **Runner** is a special test machine, belonging to you and running special software, that Gitlab-CI will use to run your actions. We'll see how to set up and customize the Runner later in this tutorial. In particular, we'll need to ensure that the \$HOME/c-tools directory exists with the right contents.
- Note: If you see the Build marked as Failed, not Pending, at this stage, it probably means there are syntax errors in the YML file these are displayed on the Build page. Fix the YML file, commit it and push it up and check the Build status again.

- The Runner machine could be a physical machine, or a VM, or a docker container. We're going to create a VM on the DoC private cloud.
- To create the Runner VM, log onto the runner-creator.doc.ic.ac.uk web interface and create a new VM based on the Featured **Non-CSG Ubuntu 14.04 30GB disk** template, which comes with college authentication and local root console access, but which does not mount DoC/College home dirs, or run the CSG maintenance system.
- All you need to do is enter your DoC login, password, and a short vm name. Please note, the vm name is only a name in cloudstack and on the vm (/etc/hosts and /etc/hostname). The name is not registered in the DNS database, and so will not resolve.

- 4 同 6 4 日 6 4 日 6

- There is quite a bit of asynchronous communication happening in the background. If you get a 502 NGINX Bad Gateway error, please check cloudstack (https://cloudstack.doc.ic.ac.uk). There is a good chance that the front end timed out before the application was completed. Your vm was probably created.
- You can ssh into the vm either by it's IP address, or by going to cloud-vm-(subnet)-(last byte) where subnet is the third byte, and last byte is the fourth byte of the address.
 Example, 146.169.46.65 would be cloud-vm-46-65.
- Check that you have sudo access:

id

You should see a reference to the sudo group.

Then you can become su:

sudo -s

- Do the rest of the setup as root via the ssh session.
- The package gitlab-ci-multi-runner is installed in the template. To connect Gitlab and our new runner VM together, register the runner:

gitlab-ci-multi-runner register

- Do the rest of the setup as root via the ssh session.
- The package gitlab-ci-multi-runner is installed in the template. To connect Gitlab and our new runner VM together, register the runner:

gitlab-ci-multi-runner register

- This asks us a few questions, first we enter the name of our Gitlab server's CI endpoint: https://gitlab.doc.ic.ac.uk/ci
- Then we copy and paste in our repo's Gitlab CI token (from the Gitlab **Project Settings** page).
- Then we enter a name for the runner I chose my-gitlab-ci-runner-ubuntu14.04-cvm - and enter zero or more symbolic tags - I entered none.
- Finally we choose the *Shell Executor* I entered shell.

くほと くほと くほと

 As soon as we have finished registering the multi-runner, go back to the Gitlab web interface, and check the Build status.
 Remember, up to now, the Build status has been *Pending*, because with no runner Gitlab can't run any tests.

(D) (P) (P) (P) (P)

- As soon as we have finished registering the multi-runner, go back to the Gitlab web interface, and check the Build status.
 Remember, up to now, the Build status has been *Pending*, because with no runner Gitlab can't run any tests.
- But now, the Build status should change in a few seconds from Pending to Failed, and the log panel (black background) should show the commands it ran and the results.
- You should see the Build successfully clone the repo on the runner VM, then fail at the first sudo apt-get command.

- As soon as we have finished registering the multi-runner, go back to the Gitlab web interface, and check the Build status. Remember, up to now, the Build status has been *Pending*, because with no runner Gitlab can't run any tests.
- But now, the Build status should change in a few seconds from Pending to Failed, and the log panel (black background) should show the commands it ran and the results.
- You should see the Build successfully clone the repo on the runner VM, then fail at the first sudo apt-get command.
- Why? After some investigation, I realised that the build runs as a local **gitlab-runner** user on the runner VM, and that user cannot use sudo by default. To discover this, I added whoami as an extra command in the **before_script** section in the YML file, then re-committed and re-pushed it:

```
vi .gitlab-ci.yml [added "- whoami" as 1st command in before_script list]
git commit .gitlab-ci.yml
git push
```

イロト 不得 トイヨト イヨト 二日

Retry the Build

• So, to allow our runner to use sudo, add the **gitlab-runner** user to the sudo group, in the root ssh session:

usermod -G sudo gitlab-runner

• Now, click **Retry Build**. If you got it right, you should see that the runner VM installs gcc and make, as the **before_script** section told it to.

- **(())) (())) ())**

• So, to allow our runner to use sudo, add the **gitlab-runner** user to the sudo group, in the root ssh session:

usermod -G sudo gitlab-runner

- Now, click **Retry Build**. If you got it right, you should see that the runner VM installs gcc and make, as the **before_script** section told it to.
- Of course, every change made to the runner VM either by the **before_script** section, or done manually as root on the VM, persists forever. Hence, once we've successfully installed gcc and make, we probably don't want to leave the apt-get commands live, because they run every time and slow things down. So comment most of the **before_script** section out (by another vi; commit and push sequence):

```
before_script:
#- whoami
#- sudo apt-get update -qq && sudo apt-get install -y -qq gcc make
- which gcc
```

• Of course, we could have avoided the **before_script** section, and run the apt-get commands manually in the root ssh shell.

1st Feb 2016 10 / 15

・ロト ・ 同ト ・ ヨト ・ ヨト

• After completing the **before_script** section, the build process attempted the **runtests** section, which reads:

runtests:

script:

- # compile it up
- export TOOLDIR=\$HOME/c-tools
- export ARCH=x86_64
- make
- # and run the tests..
- make test

• After completing the **before_script** section, the build process attempted the **runtests** section, which reads:

runtests:

script:

- # compile it up
- export TOOLDIR=\$HOME/c-tools
- export ARCH=x86_64
- make
- # and run the tests..
- make test
- When I did this, the Build process cd'd into the correct directory, set the above environment variables, and then ran make.

イロト 不得下 イヨト イヨト 二日

11 / 15

• After completing the **before_script** section, the build process attempted the **runtests** section, which reads:

runtests:

script:

- # compile it up
- export TOOLDIR=\$HOME/c-tools
- export ARCH=x86_64
- make
- # and run the tests..
- make test
- When I did this, the Build process cd'd into the correct directory, set the above environment variables, and then ran make.
- At this point, of course, \$HOME/c-tools did not exist on the runner VM, so the make failed to find **mem.h** anywhere.
- We could add rules to the before_script section to fetch a c-tools tarball from somewhere and extract it, but it's simpler to do this from the root ssh session.
- First, as you on a DoC workstation, build a c-tools.tgz tarball containing your ~/c-tools directory:

11 / 15

```
cd
tar czf /tmp/c-tools.tgz c-tools
```

• Then copy the tarball to the runner VM. I did:

scp /tmp/c-tools.tgz cloud-vm-46-64:/tmp

• Then, in the runner VM root session:

cd /home/gitlab-runner tar xf /tmp/c-tools.tgz

イロト イポト イヨト イヨト

- Then copy the tarball to the runner VM. I did: scp /tmp/c-tools.tgz cloud-vm-46-64:/tmp
- Then, in the runner VM root session: ^{cd} /home/gitlab-runner tar xf /tmp/c-tools.tgz
- Now, click Retry Build you should see that the runner VM successfully compiles and links testmld, and then runs it, producing the output:

```
basic members: { a,b,c,d,e,ldk,dcw,gnb, }
lists initially:
two: basic: { a,d,ldk,dcw, }
one: basic: { b,c, }, non-basic: { two, }
three: basic: { e, }, non-basic: { two, ne, }
T allbasic( one: b,c, nonbasic two, ): is false: ok
T allbasic( two: a,d,ldk,dcw, ): is true: ok
T allbasic( one: a,b,c,d,ldk,dcw, ): is true: ok
T allbasic( two: a,d,ldk,dcw, ): is true: ok
T allbasic( two: a,b,c,d,ldk,dcw, ): is true: ok
T allbasic( two: a,b,c,d,e,ldk,dcw, ): is true: ok
```

Build succeeded.

• Note that the Build process fetches the output from the runner and presents it to us unaltered, recording it for posterity.

- Our Build is now for the first time successful. Clicking back on Gitlab's Builds menu item shows the whole Build history, many failures plus one successful build.
- Now, every time you make any change to your repo, and push it up to Gitlab, another Build will automatically happen, and a few seconds later Gitlab's **Builds** section will show whether the new version ran the tests successfully try this a few times.
- You may well want to invest a little time wrapping your test runs in some small test harness that summarises (as Perl's **prove** does) your test runs, in order to make it clearer how many tests there were, and how many failed.
- Some languages will have their own test framework you should use, but in our case you will notice that the output comprises informational messages interleaved with test success/failure messages, the latter marked with a "T" prefix.
- As a first step, change the **Makefile** test invocation to:

```
test: testmld
   ./testmld | grep '^T '
```

・ 同 ト ・ ヨ ト ・ ヨ ト …

• Then git commit Makefile and git push the change up. In a few seconds, the latest Build output will only show:

```
T allbasic( one: b,c, nonbasic two, ): is false: ok
T allbasic( two: a,d,ldk,dcw, ): is true: ok
T allbasic( twre: e, nonbasic two,one, ): is false: ok
T allbasic( one: a,b,c,d,ldk,dcw, ): is true: ok
T allbasic( twre: e, nonbasic two,one, ): is false: ok
T allbasic( twre: e, nonbasic two,one, ): is false: ok
T allbasic( one: a,b,c,d,ldk,dcw, ): is true: ok
T allbasic( twre: a,d,e,ldk,dcw, ,onbasic one, ): is false: ok
T allbasic( twre: a,d,e,ldk,dcw, nonbasic one, ): is false: ok
T allbasic( one: a,b,c,d,ldk,dcw, ,onbasic one, ): is false: ok
T allbasic( one: a,d,e,ldk,dcw, ): is true: ok
T allbasic( twre: a,d,e,ldk,dcw, ): is true: ok
T allbasic( twre: a,b,c,d,e,ldk,dcw, ): is true: ok
```

• A simple Perl script can be used to produce more **prove** like output; I've provided one for you - see the **summarisetests** script already present in **mini-list-flattener**. To use it, change the Makefile invocation of **testmld** to read:

```
test: testmld
    ./testmld | ./summarisetests
```

- Test it yourself via make test to familiarise yourself with the summarised output.
- Then git commit Makefile and git push the change up.

Conclusion

• In a few seconds, the latest Build output will be summarised to:

```
12 tests: all 12 pass
passes:
allbasic( one: b,c, nonbasic two, ): is false
allbasic( two: a,d,ldk,dcw, ): is true
allbasic( three: e, nonbasic two,one, ): is false
allbasic( two: a,d,ldk,dcw, ): is true
allbasic( two: a,d,ldk,dcw, ): is true
allbasic( three: e, nonbasic two,one, ): is false
allbasic( three: e, nonbasic two,one, ): is false
allbasic( three: a,d,ldk,dcw, ): is true
allbasic( two: a,d,ldk,dcw, ): is true
allbasic( three: a,b,c,d,e,ldk,dcw, ): is true
```

Build succeeded.

Conclusion

• In a few seconds, the latest Build output will be summarised to:

```
12 tests: all 12 pass
passes:
allbasic( one: b,c, nonbasic two, ): is false
allbasic( two: a,d,ldk,dcw, ): is true
allbasic( three: e, nonbasic two,one, ): is false
allbasic( two: a,d,ldk,dcw, ): is true
allbasic( two: a,d,ldk,dcw, ): is true
allbasic( three: a,d,e,ldk,dcw, nonbasic one, ): is false
allbasic( two: a,d,ldk,dcw, ): is true
```

Build succeeded.

• That's enough for now. In these notes, you've seen how to set up a fresh Gitlab project repository to use **Gitlab-Cl** to do automatic testing.

(日) (周) (三) (三)

Conclusion

• In a few seconds, the latest Build output will be summarised to:

```
12 tests: all 12 pass
passes:
allbasic( one: b,c, nonbasic two, ): is false
allbasic( two: a,d,ldk,dcw, ): is true
allbasic( three: e, nonbasic two,one, ): is false
allbasic( one: a,b,c,d,ldk,dcw, ): is true
allbasic( two: a,d,ldk,dcw, ): is true
allbasic( three: e, nonbasic two,one, ): is false
allbasic( three: e, nonbasic two,one, ): is false
allbasic( two: a,d,ldk,dcw, ): is true
allbasic( two: a,d,ldk,dcw, ): is true
allbasic( three: a,d,e,ldk,dcw, nonbasic one, ): is false
allbasic( two: a,d,ldk,dcw, ): is true
allbasic( two: a,d,ldk,dcw, ): is true
allbasic( three: a,b,c,d,e,ldk,dcw, ): is true
```

Build succeeded.

- That's enough for now. In these notes, you've seen how to set up a fresh Gitlab project repository to use **Gitlab-Cl** to do automatic testing.
- Note that **gitlab-ci-multi-runner** can be used for testing several of your Gitlab projects. Set up Gitlab-CI for a second Gitlab project repo (as before) and then, on your existing runner VM, just rerun the registration using the second project's CI Token.