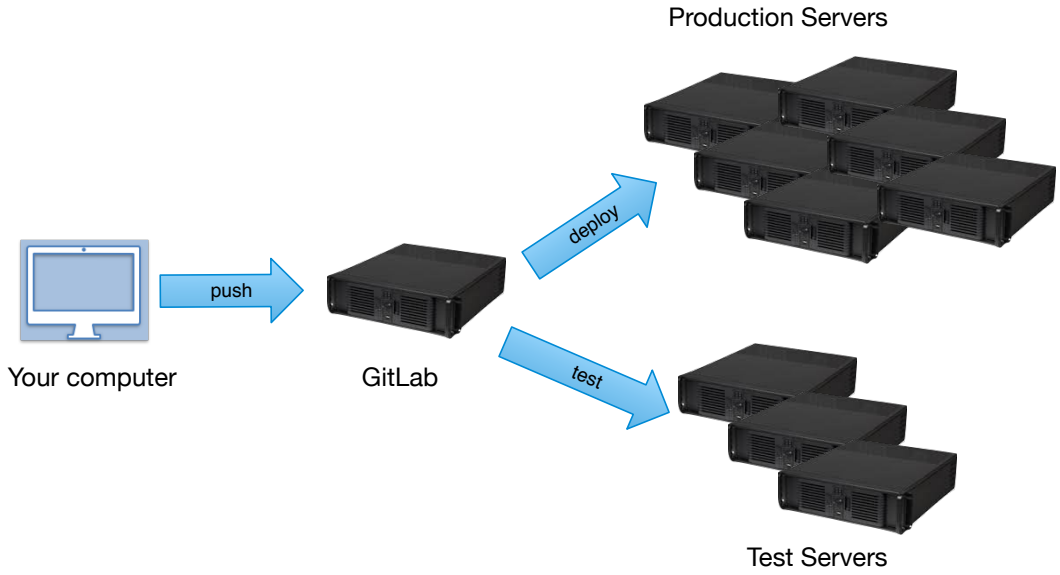
Duke Gitlab CI/CD Walkthrough
ECE 651

This is a walkthrough on setting up CI/CD on Duke's Gitlab. We assume that you already have an Ubuntu VM from Duke prior to starting this tutorial. Please note that we assume the VM is Ubuntu. If you use a verion of Ubuntu older than 20, things are likely not to work. If you do not already have an Ubuntu VM, please obtain one via `vcm.duke.edu`. Note that you need to do this on a VM that can accept network connections with a stable IP address and/or hostname, which means probably not your desktop or laptop.

1 Overview

Recall from class that in Continuous Integration/Continuous Deployment, we want to automate testing and deployment when we push to particular branches of our GitLab repository. Recall from class that the general picture looks like this:



Further recall that a daemon (in lecture: green squiggle) runs on the test and development servers to handle running of the CI/CD pipeline's jobs. You are going to setup a server (a VM from Duke's VCM) which will be both your testing and production server for this example (in reality we would want those separated). Then you are going to setup GitLab to use that server to run pipelines, and work with a small project we have put together to try that out. We'll add a few features to our CI/CD, then wrap up by talking about more things we could/should do, but are not doing here.

2 Set up VM

Gitlab's CI/CD needs a place to run your pipeline's jobs. The first thing you need to do is setup your VM to handle such jobs.

The first thing you will need is docker:

```
sudo apt install docker.io
```

Now let us set some things up for gitlab runner. Note that when you do "sudo adduser gitlabrunner" in these steps, it will ask you for a password (make it something good!) and a bunch of other information that you can just leave blank. Do these steps (note that if the first fails with "curl command not found" then you should sudo apt install curl):

```
sudo curl -L --output /usr/local/bin/gitlab-runner \
 https://gitlab-runner-downloads.s3.amazonaws.com/latest/binaries/gitlab-runner-linux-amd64
sudo chmod +x /usr/local/bin/gitlab-runner
sudo adduser gitlabrunner
sudo rm ~gitlabrunner/.bash_logout
sudo adduser gitlabrunner docker
sudo gitlab-runner install --user=gitlabrunner --working-directory=/home/gitlabrunner
sudo gitlab-runner start
```

Note that by adding the gitlabrunner user to the docker group, they have significant privileges. While some may argue that we may as well just do things as root (since having Docker access can privilege escalate), not being root helps avoid accidental catastrophes.

Leave this terminal open when you do the next step, as we will be back shortly to register the runner with gitlab.

3 Set up a project on gitlab

The first step is to create a project on gitlab. For this particular walkthrough, you are going to create a fork of the **factorserver** project. This project has a small server that reads an integer, and writes back it prime factors. You can find the factorserver project at `https://gitlab.oit.duke.edu/adh39/factorserver`. Note that you fork it with the "fork" button in the top right of the project details page.

Once you have that forked, go to Settings in the left menu, then CI/CD under it (note: there is a different top-level menu item for CI/CD—you do not want that one. You want the one under settings).

The third or fourth item down should say "Runners", hit the "Expand" button for it. On the right, find "Enable Shared Runners for this project" and click it to disabled. On the left, there should be a "Set up a specified Runner manually" section.

Now go back to your terminal on your VM and do

```
sudo gitlab-runner register
```

It will ask you for the gitlab-ci coordinator URL, put in

```
https://gitlab.oit.duke.edu/
```

then it will ask you for the registration token. Get this from the gitlab settings page you are looking at (it should be right above "Reset Registration Token" and have a little "copy to clipboard" icon right next to it). Next it will ask you for the description. You can put in any description you will recognize (like "ECE 651 CI"). Then it will ask you for the tags for this runner. In a real situation, you might have a wide variety of tags which could be where we want things to go (testing, deployment) or technologies we need available (java, postgresql, etc). We'll just use on tag here: ece651.

Last, it will ask you to enter the executor: choose **shell**. We're going to use shell, so that we can build and deploy our images easily. We'll still be using Docker for clean deployment, but we will build an explicit image in our CI commands.

If you go back to gitlab in your browser, and refresh the settings page you are on, it should now list this runner under "Runners activated for this project."

4 A brief note about debugging

Before we dive into the details of executing our CI pipeline, I just want to take a moment to talk about debugging CI pipelines. If something goes wrong with your CI pipeline, GitLab will give you the output of the failed job (click the failing job for those details). However, unless the problem is either (1) obvious from that output or (2) specific to running the job in the CI pipeline (rare), you are better off by debugging the problem locally (on your computer) rather than trying to submit more CI jobs and look at the output to figure out what is wrong.

If you have Docker installed on the computer you develop on, you can build the Docker image yourself, and run that image (and if you don't have Docker installed, we highly recommend you install it!)

If you look in .gitlab-ci.yml, you will see what we do for each stage. For the build stage, find where it says

```
build:
```

and under that you will see

```
script: docker build --build-arg LOCAL_USER_ID=‘id -u‘  --tag citest .
```

That means that to build the Docker image, we run this command:

```
docker build --build-arg LOCAL_USER_ID=‘id -u‘  --tag citest .
```

(note the . on the end is part of the command).

That means you can run that same command directly to build the Docker image (which will be named citest) on your computer.

Likewise if you look in the test: section, you will see that the script: says that we do

```
scripts/run-tests-in-docker.sh
```

So you can just run that command to run the Docker image and see what happens. This is much faster and more direct than submitting a job to the CI pipeline to debug it. But wait, there is more!

What if we want more than just the output? What if we want to explore what is going on in the Docker container to see what is setup wrong?

We can look at scripts/run-tests-in-docker.sh to see that it does:

```
mkdir coverage
docker run --rm -v `pwd`/coverage:/coverage-out  citest scripts/test.sh
```

So after we make sure we have a directory called coverage, we could do

```
docker run -it --rm -v `pwd`/coverage:/coverage-out  citest bash
```

Notice that we changed two things:

1. We added -it (-i is "interactive" and -t is "allocated a tty")
2. We changed the last argument from "scripts/test.sh" to "bash". This last argument is "what to run as the main task of the container" and we will now run bash (the command shell) allowing us to run commands in the container, as we would on any other Linux system. You can run individual commands from test.sh, ls to see what files are where, etc to try to see what is going on.

5 Executing our CI pipelines

Now that we have things setup on our GitLab project, and the server that will run the gitlabrunner daemon, it is time to go clone our gitlab repository and do development work. Go to your normal development environment (*e.g.*, VirtualBox on your laptop—NOT the server VM), and clone the gitlab project your forked earlier.

Gitlab's CI will execute our pipeline when we push changes. At the moment, all our changes are pushed. Let's make one small change to our code so that a push will do something meaningful:

Go into the `app/src/main/java/factorserver/FactorIO.java` and find the `GREETING` constant. Change it to have your name instead of mine (if your name is also "Drew" put a last initial).

Now do a gradle build (you are using Emacs, so this is just "C-c C-v", right?). You will see that it failed our test cases in `IOTest` because our prompt messages are no longer what are expected. Go to `src/test/java/factorserver/IOTest.java` and fix the `GREETING` here to match what you changed. Build again and all should be well.

Now commit and push.

Note that the first time you do this, the build job will take 3–10 minutes, because it needs to build the docker image from scratch. This process requires downloading the base Ubuntu

image, installing various packages (emacs, java, git, gradle, etc). Subsequent pipelines can re-use that image and will be much faster.

Go to the CI/CD menu item (now the top-level one, not the one under settings) in gitlab in your browser. You should have a "passed" pipeline. You'll see that under "Stages" there are two check marks—hovering your mouse over them will show you that the first is "build" and the second is "test" (which runs our JUnit tests).

Let's just see what happens if our tests fail. Go into the code, and break one of the tests (introduce any bug you want). Now push again.

The new pipeline will fail. If you click on the pipeline number (second column) it will show you more details: build succeeded, but test failed. Clicking over to the failed jobs tab will give you the output of this job.

Note that a failed pipeline may come from a variety of reasons. It is good to go look at the failed job and see what went wrong. If something is setup wrong in your pipeline, if you are missing a file, or if you code fails a test, the pipeline could fail. Looking at the output in GitLab's web interface will tell you what happened and help you diagnose the problem.

Let's go back and fix your bug before proceeding.

6 Test Coverage

We've been using Clover to determine coverage of our test cases already this semester. This CI pipeline is setup to have Clover produce an html report, and display it with gitlab pages. If you find "Settings" on the menu on the left of the gitlab webpage, and then "Pages" under that, you will get to a page that says "Congratulations! Your pages are served under:" in the middle. Clicking on that link will bring up the html version of Clover's report. This report has some other details (such as "Code metrics") but presents basically the same coverge report that you have already seen (Note: it computes coverage percent slightly differently than we do for grading, since we require branch coverage).

Let us also have gitlab report our coverage percentage on both the pipeline jobs and the project's home page. If you look at the test script, you will find that it runs `coverage_summary.sh`. This script just asks Emacs to display the coverage results and save them to a file, then greps for the Total result. The important part of this script is that it will print a line like

```
TOTAL COVERAGE: 100%
```

We want gitlab to understand that the 100 from that line is our coverage percentage. Go back to "Settings" → "CI/CD" in gitlab's menu. At the top of that page is "General pipelines." Expand that and scroll down until you see "Test coverage parsing." There is a box to enter a regular expression[1], which needs to match the line with the coverage results. For our output, the regexp would be:

```
TOTAL COVERAGE: \d+\%
```

[1]We assume most of you are familiar with regular expressions from your algorithms class. If not, the short version is that they describe patterns of text.

Unfortunately, to see this result, we still have to go into "CI/CD" → "Jobs" and then find our test job (the coverage is the last column with a name on the right).

Let's edit the README.md file (which displays on the main project page) to include links to our coverage "badges." We can also just put a direct link to the coverage details here to make it easier to find.

On the first line of the README.md file (after "ECE 651: CI/CD Intro: Factoring Server") add two image links. The first is for your "pipeline badge" and should look like this:

```
![pipeline](https://gitlab.oit.duke.edu/NETID/PROJECT/badges/master/pipeline.svg)
```

and the second like this:

```
![coverage](https://gitlab.oit.duke.edu/NETID/PROJECT/badges/master/coverage.svg?job=test)
```

Of course, you should replace NETID with your own netid, and PROJECT with your project name.

Let us also add a link to our detailed coverage information. Go down to the end and add

```
## Coverage

[Detailed coverage](https://NETID.pages.oit.duke.edu/PROJECT/dashboard.html)
```

Now, we'd like to push our changes, but we do not really want to run the entire CI Pipeline again—we haven't changed any code, or anything code related (like docker files, scripts, data, configuration, etc). So when you push, run:

```
git push -o ci.skip
```

Note that here, `-o` just passes the next option to the GitLab server, for use in its processing of the push command. GitLab sees the `ci.skip` and takes it to mean to not run any CI pipelines on this push.

7 Deploying

We would also like to deploy our prime factoring server when we push to master (and only to master!).

To change how our pipelines work, we need to edit `.gitlab-ci.yml`, and add another job:

```
deploy:
  tags:
    - ece651
  stage: deploy
  script: ./scripts/redeploy.sh
```

Note that the redeploy script is already included, so you don't need to write it. This script is pretty simplistic: it just kills the existing docker container of our server (if any) and then starts a new one. For a real redeployment, we would want to do something more complex, so that we can gracefully handle anything currently being done.

We also need to go up to `stages` and add `deploy`. Look near the top of the file, and change stages to be:

```
stages:
   - build
   - test
   - coverage
   - deploy
```

Go ahead and commit and push this, then give it a try! Wait until your pipeline completes, then give gradle a few seconds to get your server running[2]. Then you can connect with the following[3] (replace YOURSERVER with the hostname of your server—note this is the VCM server where you setup gitlab runner, *e.g.*, vcm-XXXX.vm.duke.edu):

```
netcat -N YOURSERVER 1651
```

You should get your welcome message and prompt, and be able to type a number and get it factored. Congratulations, you have successfully deployed to your server!

8 Test Our Deployment

But wait: we had to *manually* test that our deployment worked correctly. That may not sound like a big deal (after all, at this point we have built the docker image, passed our test cases, and are just running it...) but it could be—what if something goes wrong? In fact, this step is a really critical one to test, because if something went wrong and we didn't actually deploy correctly, our service is down.

What can we do? Well, lets add another stage to our CI/CD pipeline to *test our deployment*. Go back into your `.gitlab-ci.yml` and add another job:

```
test-deployment:
  tags:
    - ece651
  stage: test-deployment
  dependencies:
    - deploy
  script: ./scripts/test-deployment.sh
```

[2]The deployment script runs docker detached, so it just exits immediately. It takes a handful of seconds for gradle to start up your application inside docker.

[3]If you are not familiar with netcat, it is a program which makes network connections to other computers. The way we are using it below, will basically connect your standard input/output to a socket connected to the server's port 1651.

As before, go back to `stages` and add it there:

```
stages:
   - build
   - test
   - coverage
   - deploy
   - test-deployment
```

Note that the `test-deployment.sh` script uses netcat to make a connection, and waits until that succeeds. If it fails 20 times, it gives up (1 second per try)—this gives gradle enough time to get the server up and running. After that, it sends one request and makes sure it gets the right response.

You might think "only one test case?" but remember that at this point, we have run our entire test suite and are very confident our server works—we are just testing that it deployed correctly.

9 Further Improvements

If we were doing something real, we would want to make several other improvements to our CI/CD setup:

1. Multiple branches. We wouldn't actually do everything on `master`. We would, in fact, use `master` to deploy to production. We would likely have `testing` (or some other name for the same purpose) for code that we think is ready to go, but not yet deployed to production.

2. Separate servers for testing and production. We would setup our CI pipleine to deploy `testing` to the test server and `master` to the production servers. Note that you use the `only` keyword in your `.gitlab-ci.yml` file to make things happen only on specific branches.

3. We would want our `test-deployment` to connect to our deployed instance from a *different* machine. Doing it from the same machine means we are not testing a variety of things that can go wrong (*e.g.*, if our firewall blocks that port from the outside...).

4. A smoother handover of requests from the old to the new. At present, we just kill the old one, dropping any requests that it might be handling. Then we start a new one. Ideally, we would start the new version, test that it is deployed correctly, *then* switch new requests to go to it (while letting old requests complete). Finally, once the old version is idle, we would terminate it. Note that this functionality would be another stage or two in our CI/CD pipelines.

10 More Information

This tutorial has gotten you set up with the basics, so that you have a starting point for your own projects (both in this class and beyond).

For further reference, you might want to consult:

- The gitlab CI documentation: `https://docs.gitlab.com/ee/ci/yaml/`
- The gitlab runner documentation: `https://docs.gitlab.com/runner/`
- The Dockerfile documentation: `https://docs.docker.com/engine/reference/builder/`
- The docker run documentation: `https://docs.docker.com/engine/reference/run/`