How To Set Up CI/CD Workflows In GitLab & GitHub

usersnap

Introduction

Continuous integration and continuous deployment (also known as CI/CD) is a methodology and philosophy of how to set up your development process to include building, testing, and deployment of code automatically.

If you've worked on large complicated code projects with many team members, you probably have experienced the pain of merging code. Bugs, integration issues, new problems, and deployment issues are common occurrences.

When you create a pipeline that automatically builds, tests, and deploys every code push you make, those issues that might pile up, are brought to light immediately so you can solve these time consuming problems while they're small. This saves you and your team a lot of headaches, code refactoring, and stress.

In this guide you will learn all about what CI/CD is, what the benefits are, and how to implement a full CI/CD pipeline in GitLab and GitHub. You will also learn about tips on quality assurance and gathering user feedback for continuous product improvement.

Table of Contents

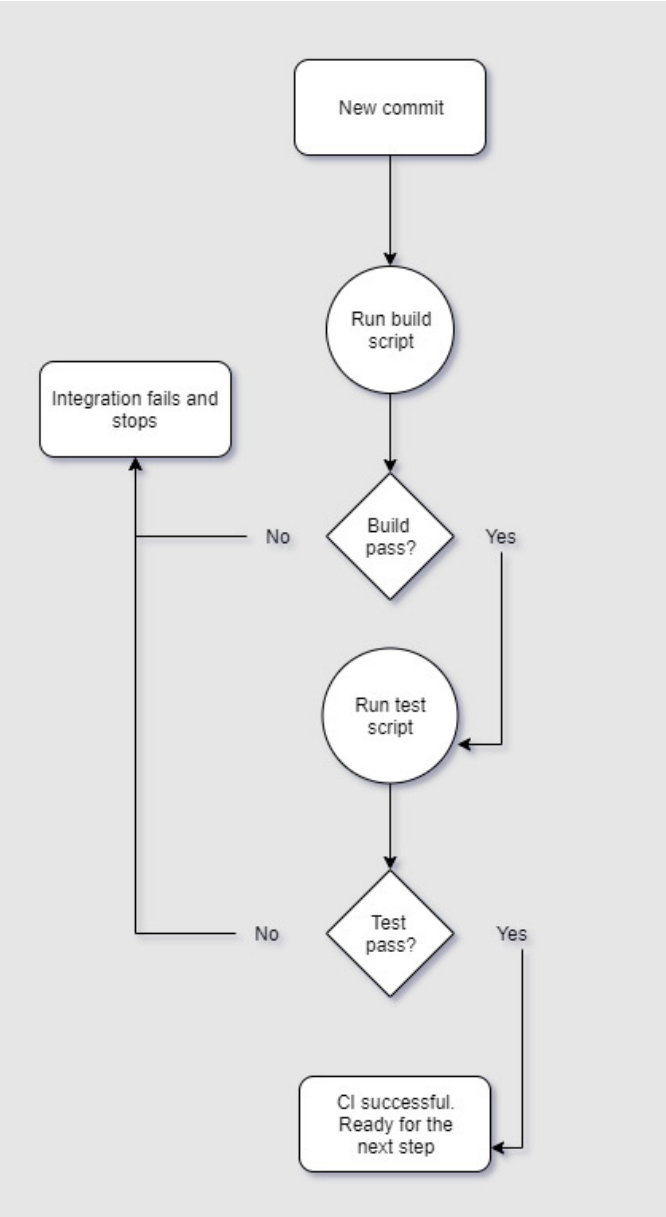
usersnap

What is CI/CD?

Continuous integration (CI) is the part of this process that integrates new commits in the repository into the main code base or branch.

This integration usually has two parts to it: building and testing.

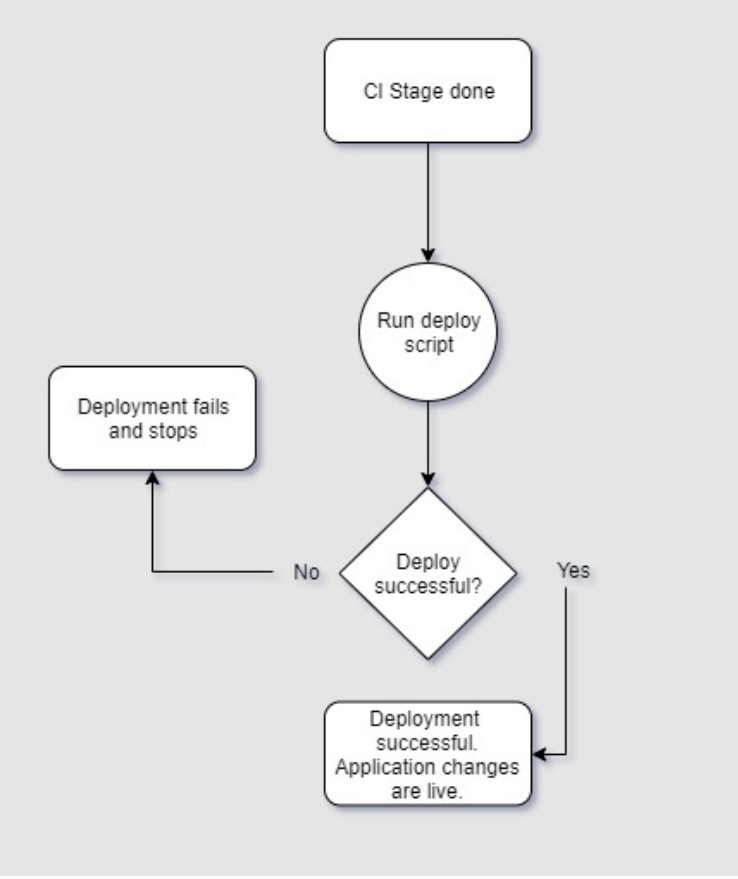
By building and testing every commit, you reduce bugs and headaches involved with making changes to a large code base with multiple contributors.



usersnap

Continuous deployment or delivery is the CD part of this process and they're different but similar.

Continuous deployment is when you take your new commits to the repository and push those changes live onto your production server. By continuously deploying new changes in your application, a potential buildup of bugs and errors is better managed. Rather than deploying many changes at once and seeing the errors associated with those changes, you deploy small changes often which brings the errors to light faster.



Continuous delivery is the step before deployment. After the CI stage happens, the code base should be ready to be deployed. Continuous delivery takes the most recent changes and prepares the code to be deployed on a new branch to be ready for deployment.

What is a pipeline?

A pipeline is how to describe the set of stages set in your CI/CD configuration.

If your CI/CD has 3 stages: build, test, deploy, then your pipeline will look like:

A CI/CD pipeline is the end to end steps your repository takes when new commits are pushed to the repository.

Why is CI/CD important?

Creating an app for people is hard. Getting users to use your app is hard.

The last thing you want if you get users to use your app is for it to break or have an error.

According to testlio, less than 25% of users return after 1 day and the average app loses 95% of users after 90 days.

This means you don’t have many chances to make a lasting impression and by adding tests to your code you can ensure that you’re providing a perfect experience for your users.

The difference between top apps and forgotten apps is the rate of deployment and delivery. According to the 2016 State of DevOps Report, the difference between high performing product teams and low performing teams is the frequency of deployments.

The top product teams deployed up to 200x more often than low performing teams.

If you implement a CI/CD pipeline in your project, deploying new changes to your app is as easy as making a new commit and push.

Just as important as a robust development process, you also want live user feedback to inform you if the updates and new features are working well.

It is important to have a light-weight visual feedback tool in your application to gather bug reports and usability reviews to ensure users are happy with your new deployments. We will talk more about this later on in this ebook.

Adding tests and user feedback into your development process

Should you include manual testing throughout your development process? The short answer is yes.

Software development is led by people, and people are imperfect by nature. We often rush our code, or see if the code compiles and then consider it done. When you add checks in place such as unit or integration tests, the likelihood of creating serious issues in your software gets reduced dramatically.

Still not convinced? Here are some horror stories caused by software bugs.

In 2010, Mt. Gox, a Japanese Bitcoin exchange, was hacked and over 850,000 Bitcoins were lost. Today the value of 850,000 Bitcoin is equal to over 6.6 billion USD.

The CEO of Mt. Gox was quoted as saying:

> *We had weaknesses in our system, and our bitcoins vanished.*

Here's another one.

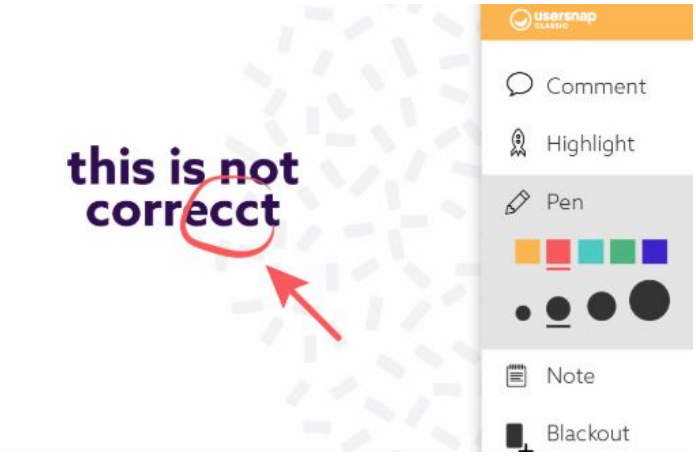
A system involving luggage at Heathrow Airport was incapable of handling luggage in real world situations. This failure of testing in real world situations led to 42,000 bags not traveling with their owners, and over 500 flights being canceled in the span of 10 days.

The issue at Heathrow Airport points to a major issue most software projects don’t take into consideration at the start of their project. That issue is getting real world user feedback in real world situations.

If users are able to report problems and give feedback instantly to software managers, such disasters could have been controlled.

This is a problem that Usersnap has built a solution for.

Usersnap makes it very easy for users visiting your website or on your app to provide feedback in a visual way - screenshots, drawings and comment boxes.



Most of your company users probably won't know how to raise an issue in GitHub or GitLab, and your customers definitely don't have access to them. But every user and customer can be a source of amazing insight.

By integrating Usersnap with GitHub or GitLab, your customers or users can send instant visual feedback to your GitHub or GitLab's issue tab.

Implementing CI/CD pipeline

Now you will learn how to implement a CI/CD pipeline of your own.

To be successful in these tutorials, there are a few things you need to have/understand:

1. An understanding of git, its commands and how GitHub and GitLab work.
2. An AWS account and understanding of how to use it.
3. Elastic Beanstalk and IAM accounts.

One last thing, the sample API project we will be using in these tutorials can be found here.

GitLab CI/CD Deployment to AWS Elastic Beanstalk

In this section we will go over a step by step guide to setting up your project with CI/CD in GitLab and deploying that project to an Elastic Beanstalk environment on AWS.

The sample project this tutorial will be using can be found here.

To deploy to AWS Elastic Beanstalk you must have an AWS account. You need an AWS Access Key Id and an AWS Secret Access Key for the deployment part of this project.

If you don’t want to create an AWS account you can still follow along with this guide, just skip the deployment part (at page 29).

The process

Create a simple CI/CD pipeline
↓
Make echo statements appear in the GitLab pipeline
↓
Add the sample project
↓
Add the build and test jobs configuration
↓
Set up Elastic Beanstalk environment
↓
Add the deploy configuration
↓
Review

Create a simple CI/CD pipeline in GitLab

First you need to create a GitLab repository to work with. After your GitLab repository is ready, you can continue with creating a .gitlab-ci.yml file.

The main requirement of creating a CI/CD pipeline in GitLab is the ".gitlab-ci.yml" file.

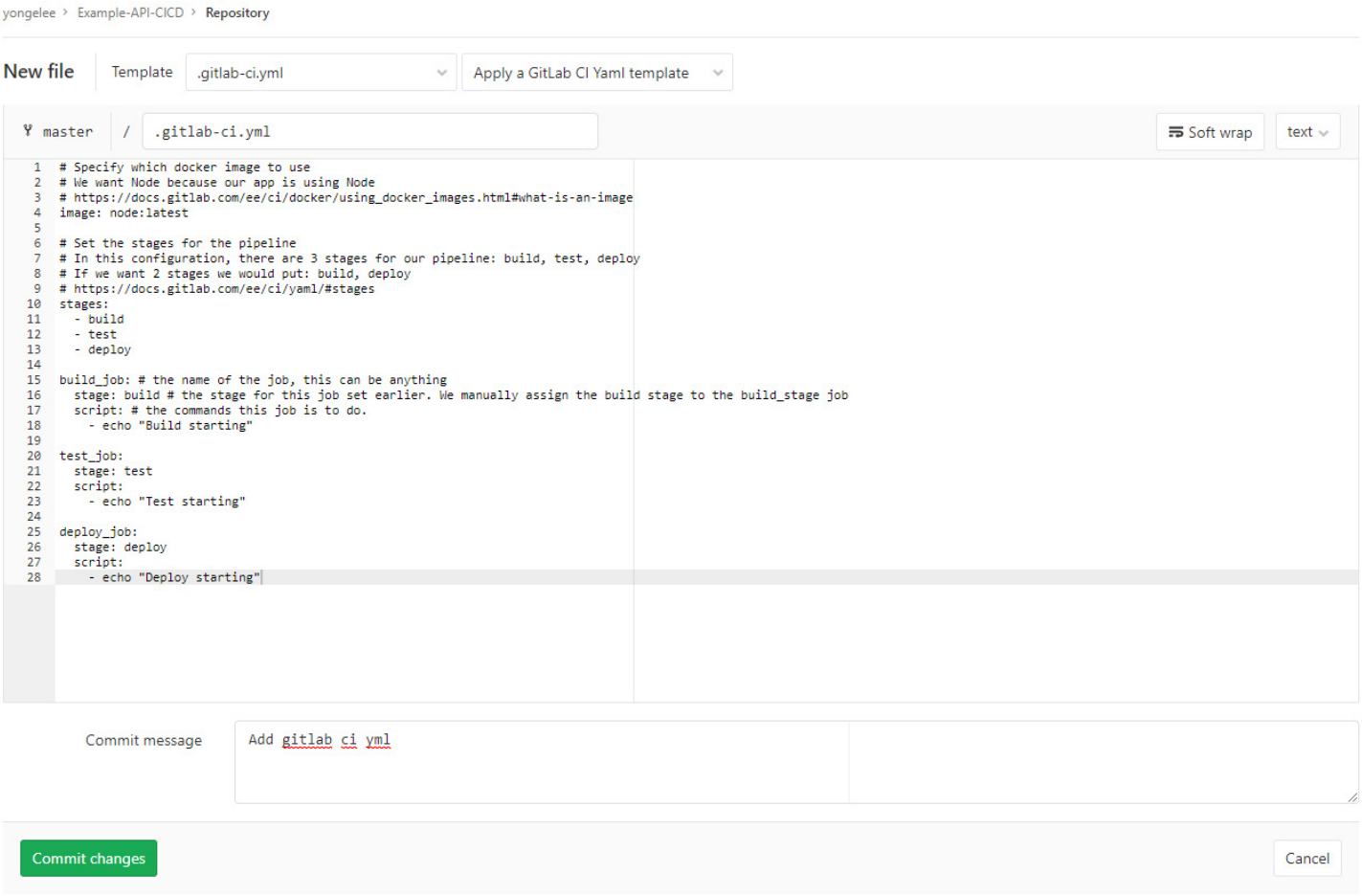
This is the file that GitLab looks for to set up its pipeline. Let's create a very simple pipeline that has 3 stages: build, test, deploy.

In these stages, let's print out some statements so we can see how our configuration file works.

In your GitLab project page, click the "New file" button to create the .gitlab-ci.yml file.

Here is a gist for you to copy.

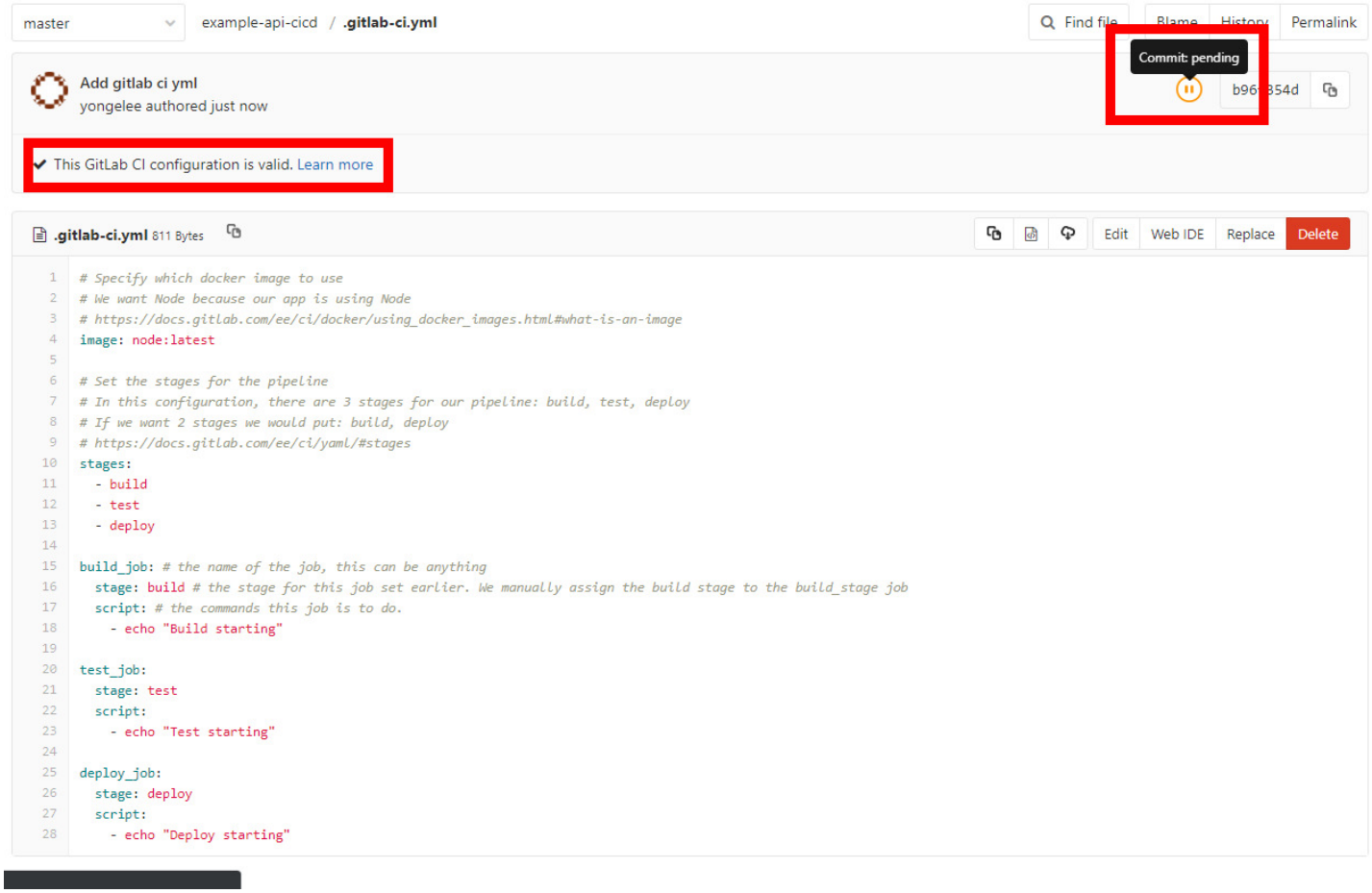
Your screen should look like:



In this .gitlab-ci.yml file we are doing the following:

1. Setting the image to be a Node.js environment
2. Defining the stages of our pipeline (build, test, deploy)
3. Creating jobs, assigning stages to each job, and creating scripts for each job.

After we add this file to the GitLab repo, we've essentially created a pipeline. Let's commit this file and see what happens. Press the "Commit changes" button.



You should now see this page. Notice the "This GitLab CI configuration is valid" text in the header area. This means the yaml file we created is valid and GitLab can use it.

Also in the top right corner there is a pause icon and when you hover over it you see the "commit: pending" text. This means the pipeline has started.

Make echo statements appear in the GitLab pipeline

Now that we've added our .gitlab-ci.yml file to our project, GitLab will see this file and set up a pipeline for us.

In this pipeline we've created 3 stages: build, test, deploy.

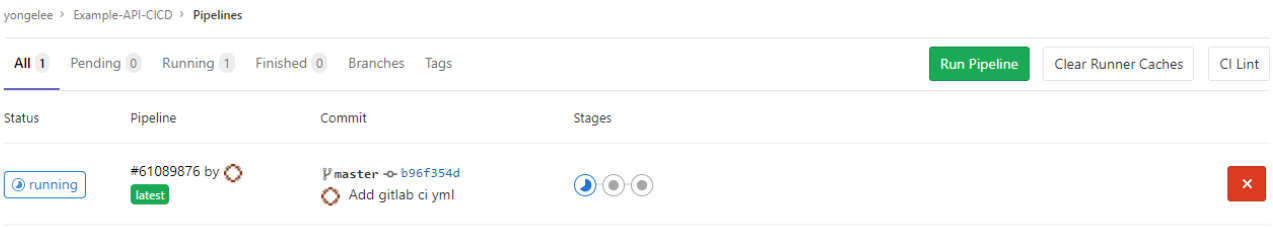
In each of these stages, we're only printing lines to the console by using the echo command.

Let's see how the pipeline we've created works.

In the left side navigation, there is a menu item called "CI / CD" which contains all of the options for CI / CD.

Click through to the "pipelines" option.

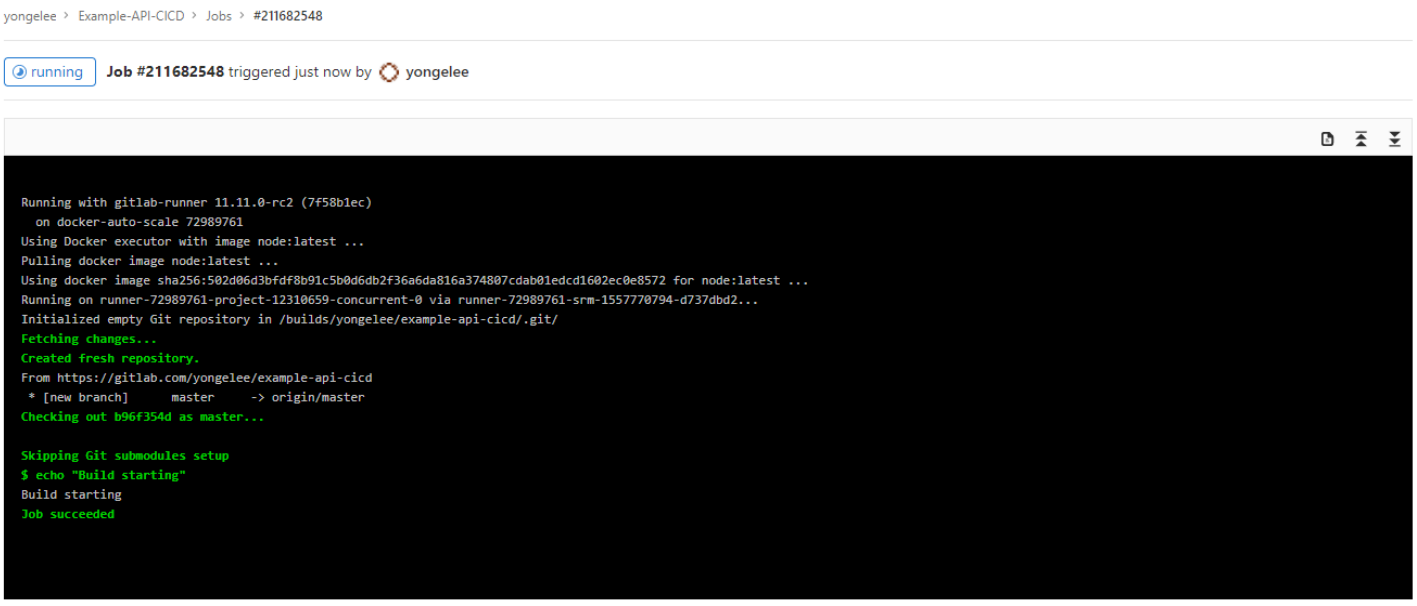
You should then see something like this:



This is our first pipeline and it's created based off the .gitlab-ci.yml file we created.

Under stages, you can see that there are 3 circles. This is because we created 3 stages in our configuration. If we set up 2 stages (build, deploy) then there would be 2 circles. If we set up 4 stages (pre build, build, post build, deploy) then there would be 4 circles.

If you click through to each job by pressing one of the circles or going to the jobs section under CI / CD in the left side navigation you'll see a screen like this:



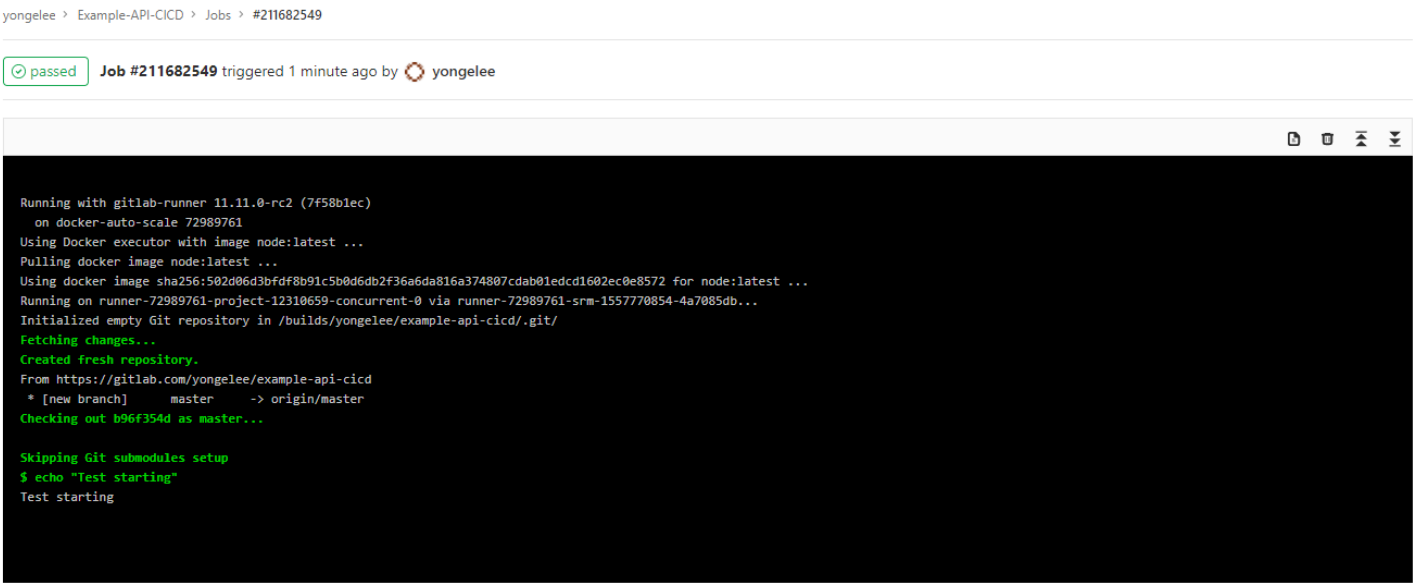
The first stage in our pipeline is “build.” In our build job we have 1 statement in our script which is `echo “Build starting”.

Imagine typing a build script manually on a terminal. You might run “npm run build” or download some packages.

The build job is meant to automate those commands. So instead of always typing “npm run build” every time you commit a change, the build job you created automatically runs these commands for you.

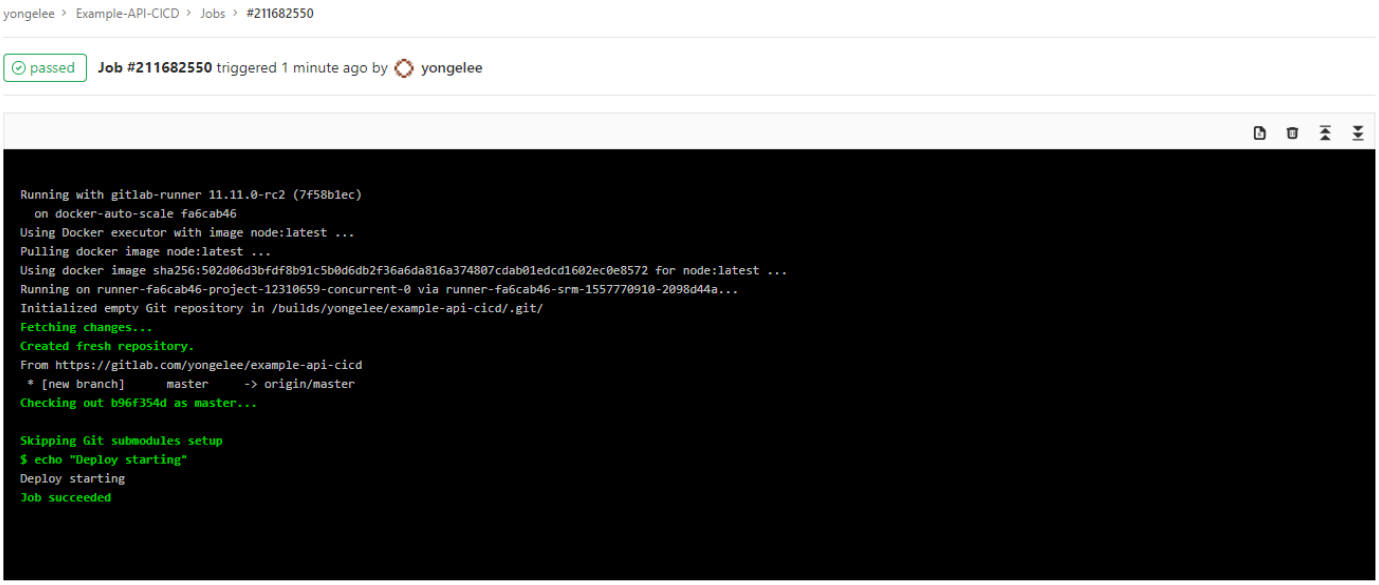
If this job fails, for example if you have a “npm run build” and for whatever reason it fails, then the next step in the pipeline won’t run and the pipeline stops.

If the job succeeds, like in this example since we just print out a message, then the pipeline moves onto the next stage which we’ve set as “test”.



In the test page, it should look the same except a different message. This time it's "Test starting."

In this stage you would run your tests through a command like "npm run test" and if the test cases pass, then it would continue onto the next stage, which in our case is "deploy."



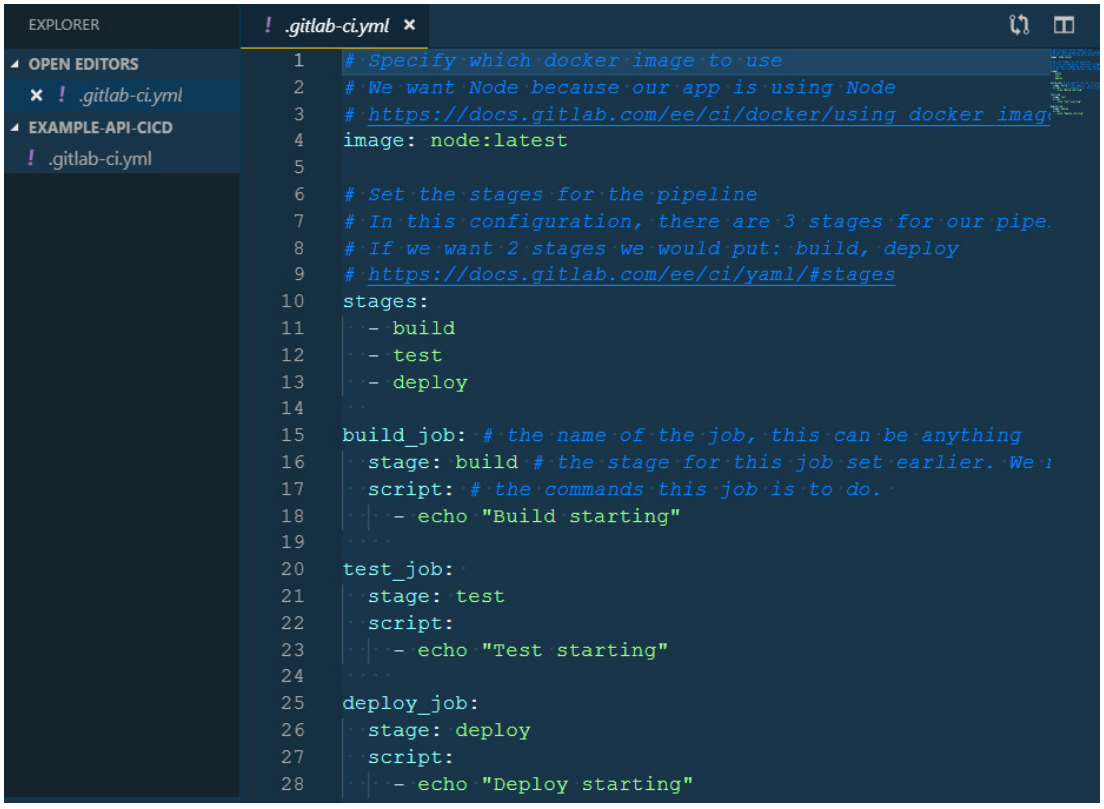
Once again, this screen is what a job looks like. It's just a console that runs the commands you set in the script.

For deploy, this is where you would do what it takes to get your project running on a production server. For us, this will be where we deploy our project to an Elastic Beanstalk environment on AWS.

Add the sample project to GitLab

We have a basic pipeline running in our GitLab repo. Next we have to turn this CI/CD template into a real application. So let's add our sample project to this repository.

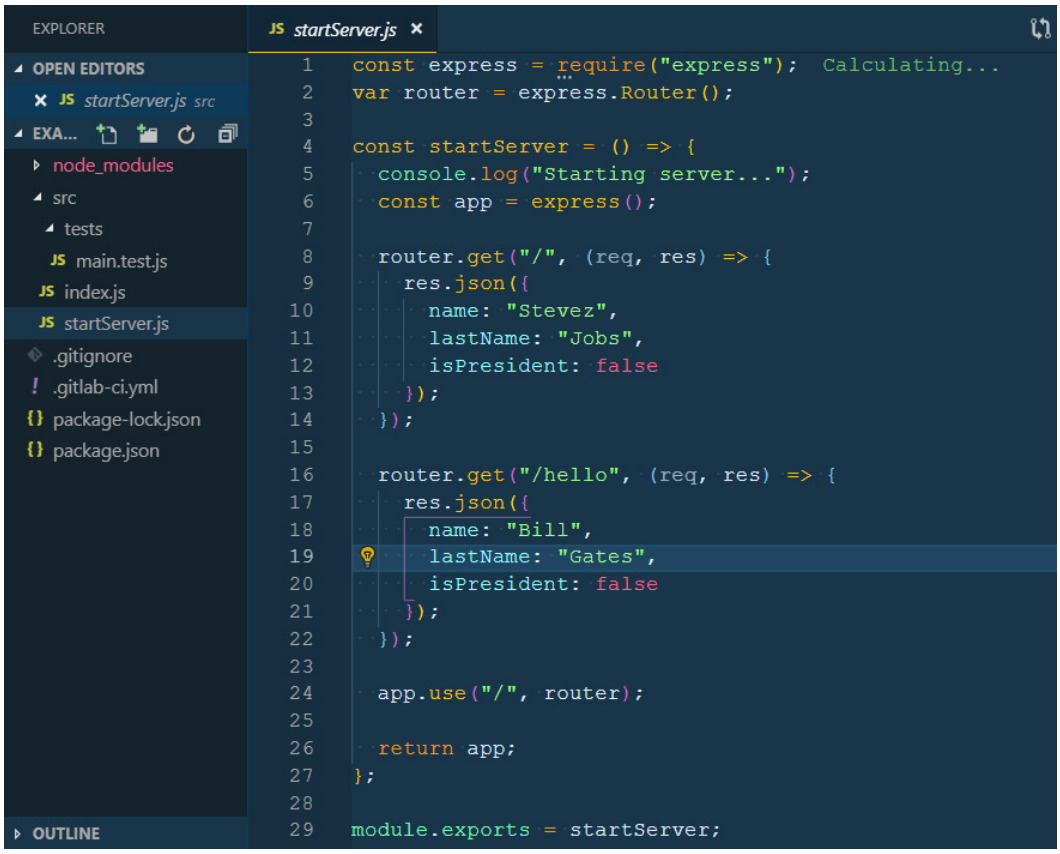
Since this was a new project and we added the .gitlab-ci.yml file through the GitLab interface, we can clone this repository on our local computer and copy and paste the sample project into this project.



Add the sample API project files to this project.

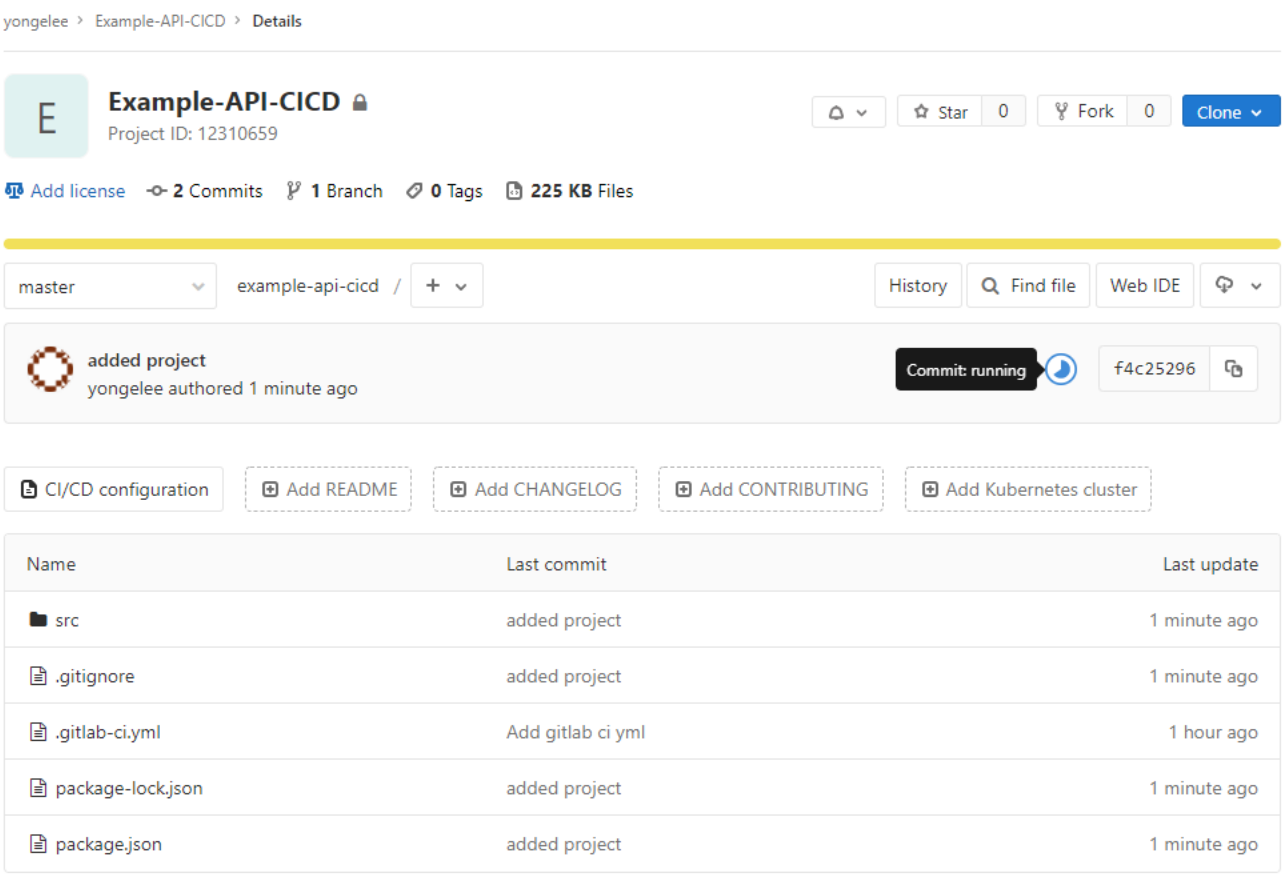
After you add the project files, your project should be the same as the sample API project except with the .gitlab-ci.yml file added.

At this point your project should look like:



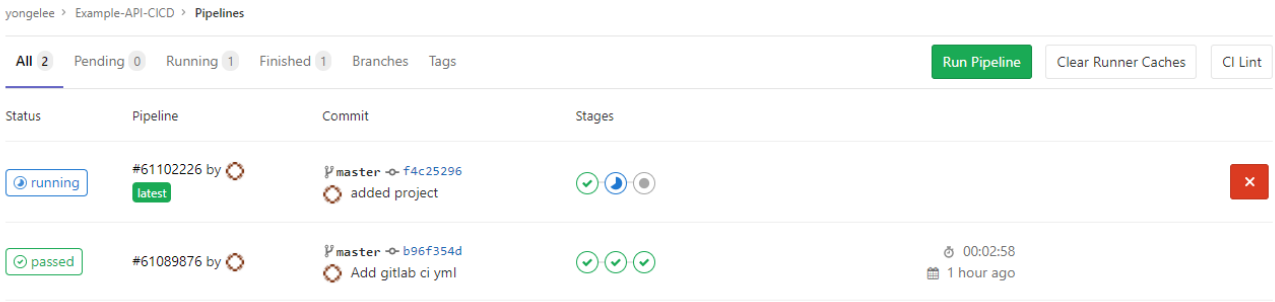
Commit and push your changes to GitLab.

Then your GitLab project will look like:



We've pushed a new change to the repo, that means the pipeline will rerun its process again.

Since this is our 2nd push with an active CI/CD configuration file, our 2nd pipeline will be created.



Add the build and test jobs configuration

We've created a GitLab project, added an API project to it, and have set up 3 stages in our pipeline.

Now let's create some realistic jobs for our build and test stages.

Build

What does the build stage do? Well it builds your code into the production version that you want to use.

For Java, C++ and some other languages, your code has to compile into something you can run. For Node.js and Javascript, you don't have to compile your code but you do need to install the dependencies you've added to your project.

So let's just run "npm install" in this stage.

We also need to make the node_modules file available to our project by specifying it as an "artifact." Check out the gist below to see what this looks like.

Test

Testing is a crucial part of the CI/CD process. If you don't have tests, then you might commit code into your main branch when it should not be there.

The testing stage will be different for every project. In our project we are using Jest as our testing tool but there are many other tools out there for many other languages and frameworks.

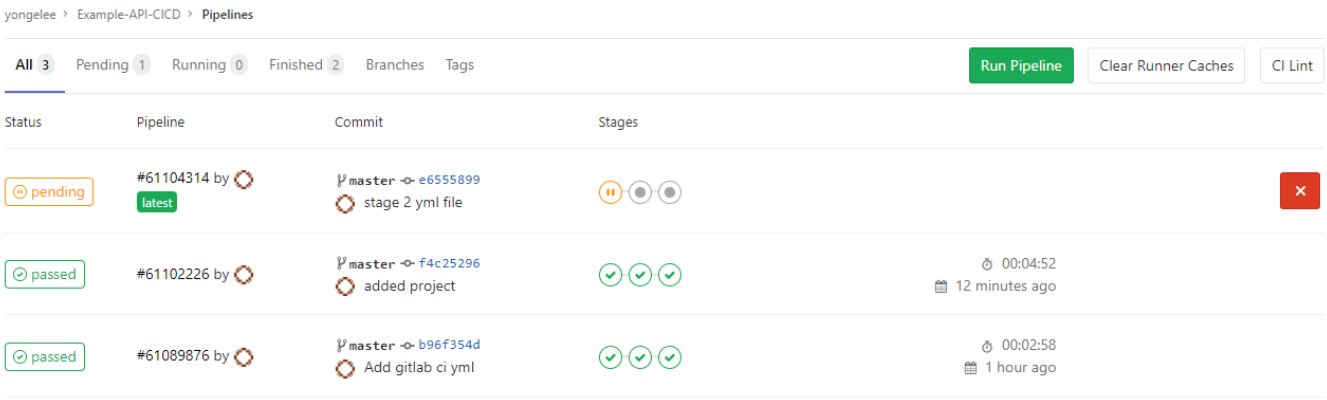
In our case, we just want to run our npm script called "test" to test our code. This will result in a success or error depending on if our tests pass.

So we need to add "npm run test" to this stage.

This .gitlab-ci.yml file is how it should look.

Commit and push the new changes to your .gitlab-ci.yml file.

Our 3rd pipeline now looks like:



Our build job now looks like:

```
Running with gitlab-runner 11.11.0-rc2 (7f58b1ec)
  on docker-auto-scale 0277ea0f
Using Docker executor with image node:latest ...
Pulling docker image node:latest ...
Using docker image sha256:502d06d3bfdf8b91c5b0d6db2f36a6da816a374807cdab01edcd1602ec0e8572 for node:latest ...
Running on runner-0277ea0f-project-12310659-concurrent-0 via runner-0277ea0f-srm-1557777504-843390bc...
Initialized empty Git repository in /builds/yongelee/example-api-cicd/.git/
Fetching changes...
Created fresh repository.
From https://gitlab.com/yongelee/example-api-cicd
 * [new branch]      master     -> origin/master
Checking out e6555899 as master...

Skipping Git submodules setup
$ echo "Build starting"
Build starting
$ npm install
npm WARN example-api-cicd@1.0.0 No description
npm WARN example-api-cicd@1.0.0 No repository field.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.9 (node_modules/fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.9: wanted {"os":"darwin","arch":"any"} (current: {"os":"linux","arch":"x64"})

added 523 packages from 366 contributors and audited 861228 packages in 12.903s
found 0 vulnerabilities

$ echo "NPM install finished."
NPM install finished.
Uploading artifacts...
node_modules/: found 7804 matching files
Uploading artifacts to coordinator... ok            id=211741183 responseStatus=201 Created token=GxFQ_gx4
Job succeeded
```

Our test job looks like:

```
Running with gitlab-runner 11.11.0-rc2 (7f58b1ec)
  on docker-auto-scale 72989761
Using Docker executor with image node:latest ...
Pulling docker image node:latest ...
Using docker image sha256:502d06d3bfdf8b91c5b0d6db2f36a6da816a374807cdab01edcd1602ec0e8572 for node:latest ...
Running on runner-72989761-project-12310659-concurrent-0 via runner-72989761-srm-1557777694-cc81b328...
Initialized empty Git repository in /builds/yongelee/example-api-cicd/.git/
Fetching changes...
Created fresh repository.
From https://gitlab.com/yongelee/example-api-cicd
 * [new branch]      master     -> origin/master
Checking out e6555899 as master...

Skipping Git submodules setup
Downloading artifacts for build_job (211741183)...
Downloading artifacts from coordinator... ok        id=211741183 responseStatus=200 OK token=GxFQ_gx4
$ echo "Test starting"
Test starting
$ npm run test

> example-api-cicd@1.0.0 test /builds/yongelee/example-api-cicd
> jest

PASS src/tests/main.test.js
  the main test
    ✓ make it run (2ms)
  Make an API call
    ✓ The call to index works (30ms)

  console.log src/startServer.js:5
    Starting server...

  console.log src/tests/main.test.js:8
    server started on port 4000

  console.log src/tests/main.test.js:26
    {"name":"Stevez","lastName":"Jobs","isPresident":false}

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.676s
Ran all test suites.
$ echo "Test complete!"
Test complete!
Job succeeded
```

As you can see, these jobs just run whatever commands we set it up to run.

Now our build and test jobs work properly. Next up is deployment to AWS!

Setting up Elastic Beanstalk in GitLab

If you know how to create an Elastic Beanstalk environment you can skip this step.

Log in to your AWS account and navigate to the Elastic Beanstalk service page. In the top right corner, press the “Create New Application” button. After that you can create a new environment.

The interface will require you to fill out some basic details about your environment. The important parts are the environment name (which you can make anything) and the platform. Our app is Node.js so we choose the Node.js preconfigured platform.

Once the environment is done building you will be able to see the main page for your environment which has all of the options you need to change in the configuration.

Now we have our environment and application setup in Elastic Beanstalk. All that’s left is to deploy it.

Adding the deploy configuration in GitLab

For this step you will need to add the Elastic Beanstalk CLI on your computer. You also need to add “named profiles” to your computer.

Check if your eb cli is working by typing:`eb --version`

If you see a message on your terminal then it’s working.

We will be using the eb cli and our AWS named profile to connect our application to the Elastic Beanstalk environment we just created.

To do this we must:

1. Initialize the project as an elastic beanstalk project
2. Set our environment to what we just created
3. Deploy it.

In your terminal, make sure you're in your project's directory.

Then type:`eb init --profile albert`
(replace albert with your profile name)

Then you will be prompted with options.

The first question is your default region. This is very important, make sure you select the region you created your environment in. To see the region you're working in, go to AWS and check the top right corner. The region is listed there.

The next question is "select an application to use." Select the one you just created.

Next is a question about CodeCommit. Since we're using GitLab we don't need that.

After this is done you must set your environment.

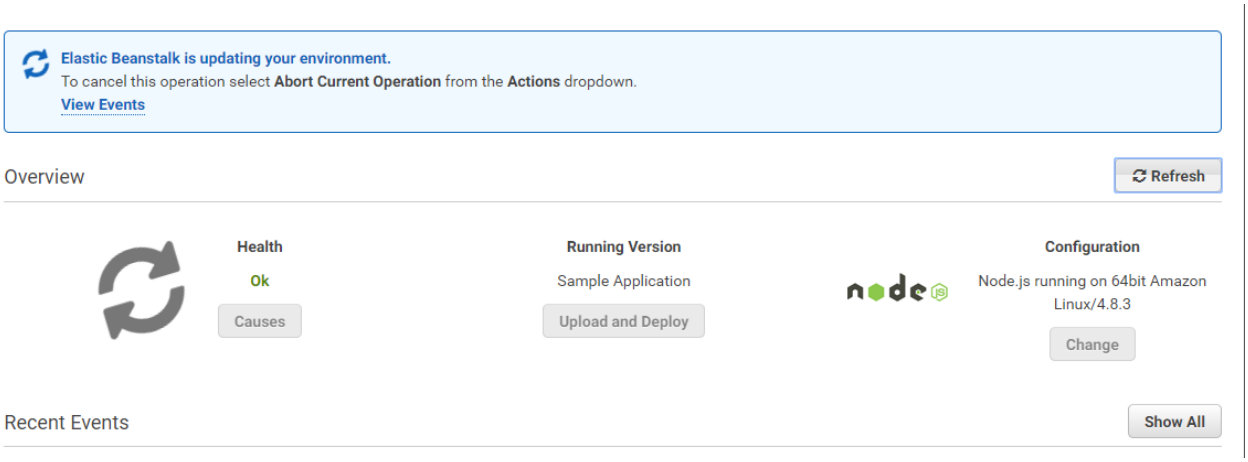
Type:`eb use cicdenv`
(replace cicdenv with the environment name you created)

Now your project is initialized as an Elastic Beanstalk project!

Let's test this out by running the deploy command.

Type:`eb deploy`

If your terminal says you've deployed ok, and you see this screen:



Congratulations, you've just deployed your app to Elastic Beanstalk!

Visit the url that Elastic Beanstalk created for you and you should see this:

```
{"name":"Stevez","lastName":"Jobs","isPresident":false}
```

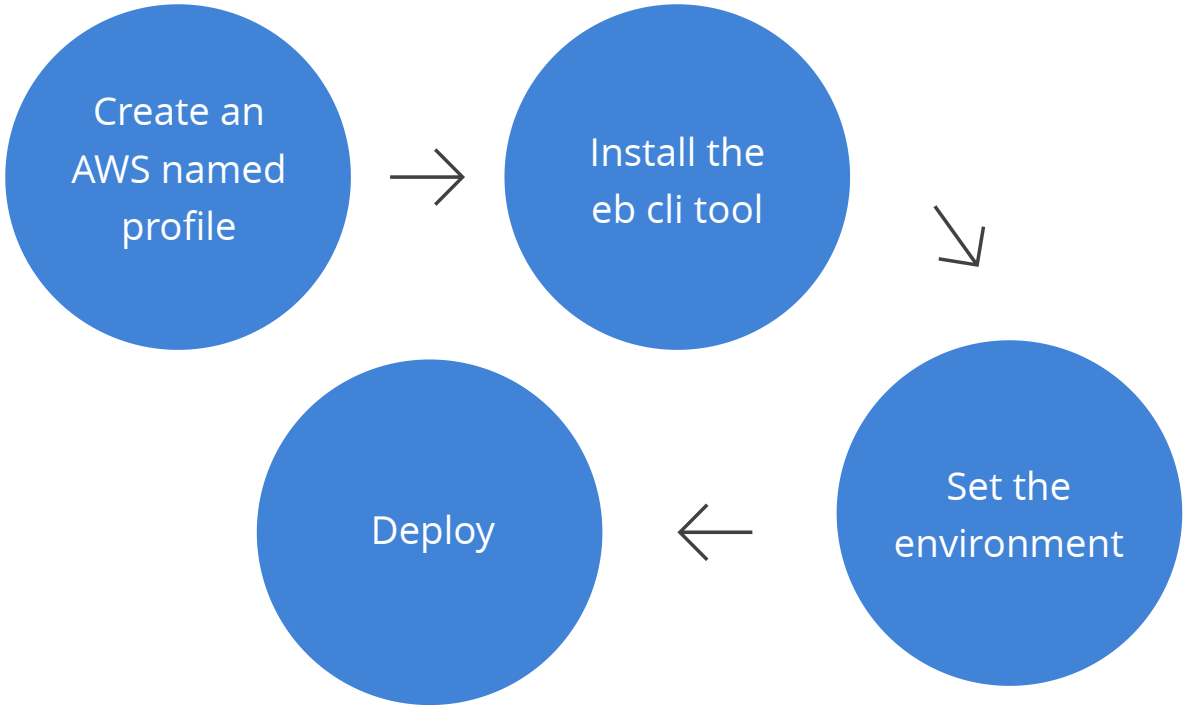
Since our API just returns a JSON object, that is all our website shows.

Great, now let's automate this process.

To automate it, we need to add our .elasticbeanstalk folder to our repo (this folder was automatically created when we ran eb init) and add the deployment scripts to our .gitlab-ci.yml configuration.

When the .elasticbeanstalk folder was created, it was added to the .gitignore file automatically. Remove the elastic beanstalk folders from your gitignore.

Let's go over the steps to deploy using the eb cli:



Let's mimic this process in the deploy job of our pipeline.

First we need to set the image of the deploy job to be python because we need pip (python package manager) to install the eb cli.

Next we create the credentials file for the AWS named profiles. Then we install the eb cli, set up our environment and run deploy.

Your .gitlab-ci.yml should now look like this.

Since we've committed our .elasticbeanstalk/config.yml file to our repository, the eb cli will know the configuration details of our environment.

And after we create a credentials file using our profile, access key, and secret key, our application will be able to authenticate to change our environment.

```
- echo -e "
[albert]\naws_access_key_id=$AWS_ACCESS_KEY_ID\naws_secret_access_key=$AWS_SECRET_ACCESS_KEY">~/.aws/credentials
```

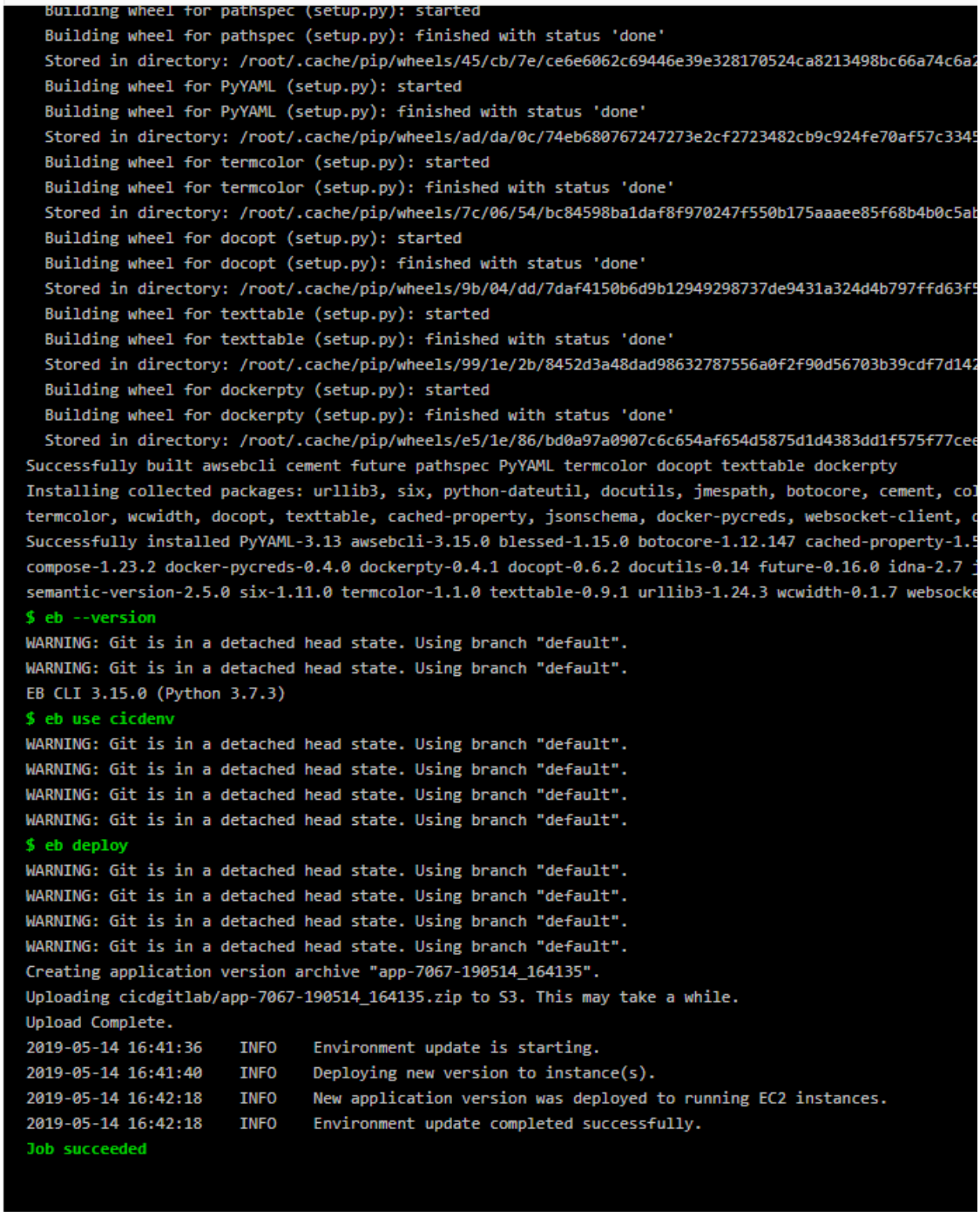
Make sure you replace [albert] with your own profile name.

This line is getting the AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY values from the GitLab environment variables.

To set your id and access keys environment variables, go to GitLab and under settings on the left side navigation, click CI / CD, then expand "Variables," then enter your secret key and id. This way your environment variables will be available in your .gitlab-ci.yml file when the pipeline runs.

Now we have everything in place for our pipeline to work as intended! Let’s push our changes to GitLab.

Our deploy job should look like this:



If you see this message that means you've deployed successfully!

Make a change to your code, for example change the name of one of the json objects you're returning and commit and push your code.

Then watch as your CI/CD pipeline runs through the whole process of building, testing, and deploying with just one push!

Review of CI/CD set up in GitLab

If you've made it this far, congratulations! You can now say you're able to implement a CI/CD pipeline with GitLab and AWS!

So to review, our steps were:

1. Create a simple CI/CD pipeline
2. Make echo statements appear in the GitLab pipeline
3. Add a project
4. Add the build and test jobs configuration
5. Set up Elastic Beanstalk environment
6. Add the deploy configuration

This is just the tip of the iceberg when it comes to CI/CD pipelines. There are many more configuration options you can use in your .gitlab-ci.yml file. Try experimenting and creating your own custom pipeline from scratch!

GitHub and Travis CI Deployment to AWS Elastic Beanstalk

In this section we will be going over how to deploy a Node.js application to AWS Elastic Beanstalk using GitHub as repository and Travis CI for our CI/CD integration.

The sample project we will be using in this tutorial can be found here.

The process

Add a .travis-ci.yml file to a GitHub repository

↓

Sign up on travis-ci.org

↓

Create a CI/CD configuration file that prints out messages

↓

Add the sample project

↓

Add the build and test jobs configuration

↓

Add the deploy configuration

↓

Review

Add a .travis-ci.yml file to a GitHub repository

Create a new repository and connect your local directory with this repository.

In your terminal, change directories to the folder you want to work in and clone the repository you just created. You should now have an empty repository and an empty folder to work with.

Create a file called ".travis.yml" this is the configuration file that Travis CI looks for to set up the CI/CD pipeline.

The travis yaml file is similar to the GitLab yaml file but takes in different configuration options. It has many configuration options but for this basic pipeline the options we care about are language, and the lifecycle methods.The language sets the environment that the pipeline will run. We are using Node so the language we will set is "node_js."

The lifecycle methods for Travis are predefined unlike GitLab. With GitLab we create our own stages and assign jobs for the stages we create. With Travis there are 12 lifecycle methods that we can use.

The ones we want to use are “install, script, deploy” but let’s create a yaml file containing many lifecycle methods and print out some messages.

Your .travis.yml file should look like this gist.

We are setting the language to be node_js with a version of 11.6.0. We set the cache as npm to tell Travis to cache the files from node_modules.

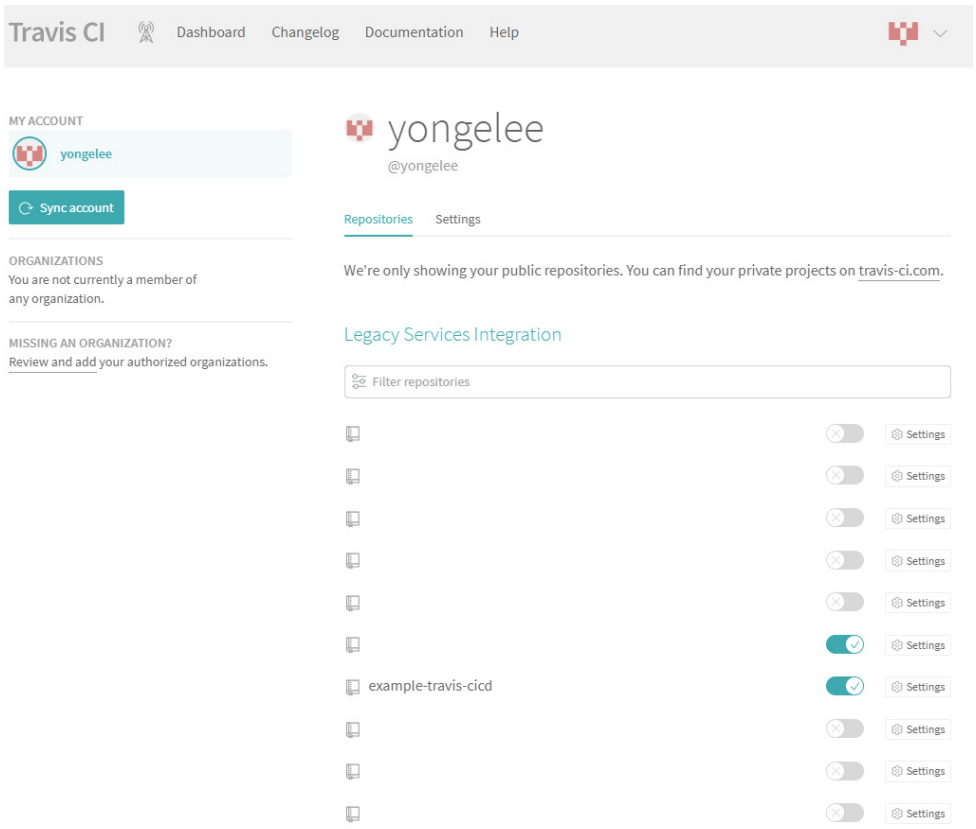
Next we print out some echo statements for the lifecycle methods to see how our configuration file works in the Travis interface.

Commit and push your changes to GitHub.

Setting up Travis CI and connecting GitHub

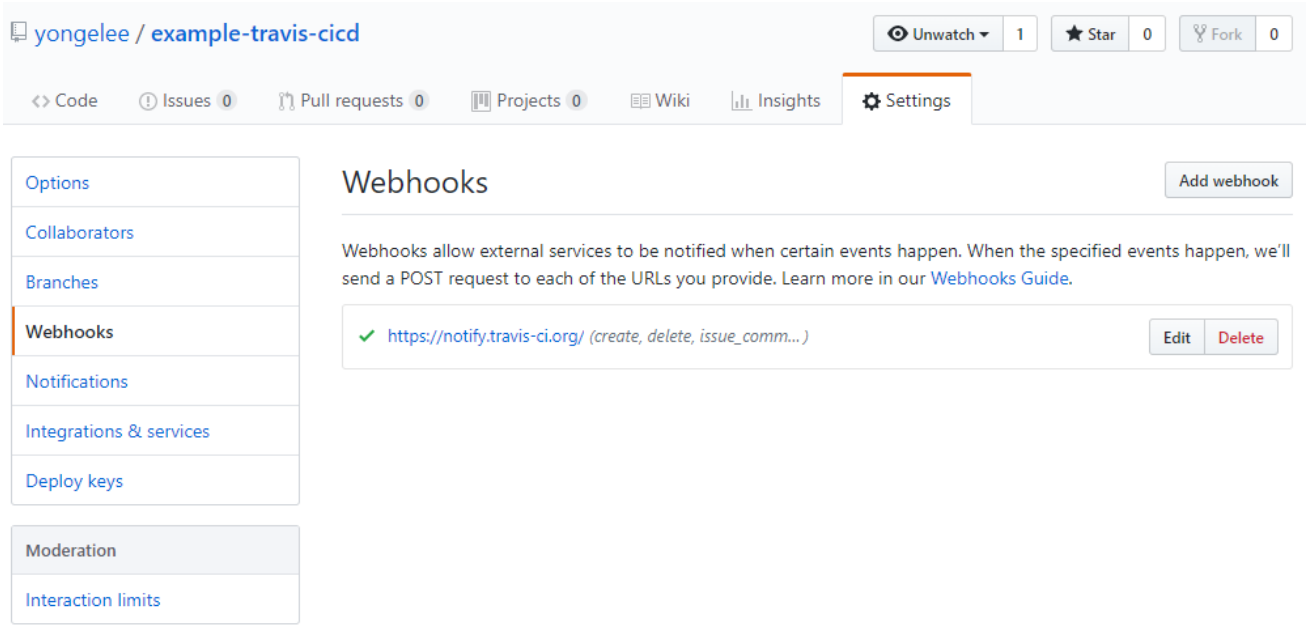
Visit travis-ci.org and create an account or sign in using your GitHub account. Activate the GitHub Apps integration, which allows Travis to have access to the GitHub repositories you want to set up CI/CD with.

After you've connected your account, navigate to `travis-ci.org/account/repositories` and you should see a screen similar to this:



Find the repository you want to work with and check the checkbox on the right side.

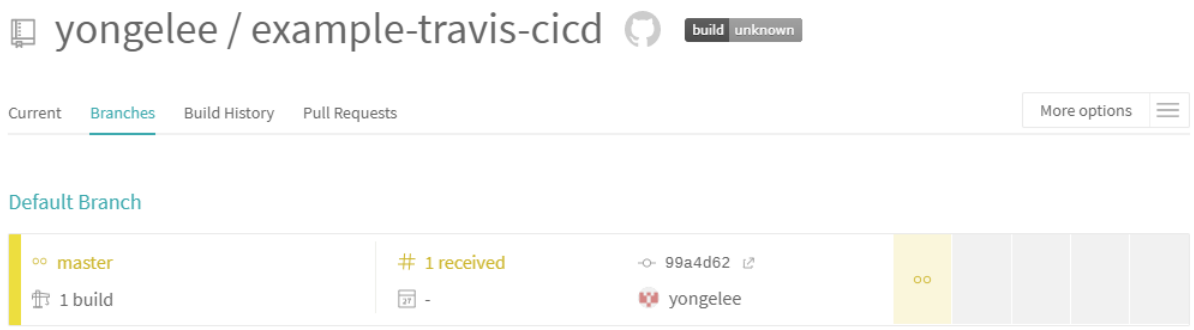
This will add a webhook to your GitHub repository that notifies Travis CI whenever there is activity in the repository.To check if this webhook has been activated, go to your GitHub repository, go to settings, and press Webhooks. You should see a screen that looks similar to this:



Create a CI/CD configuration file

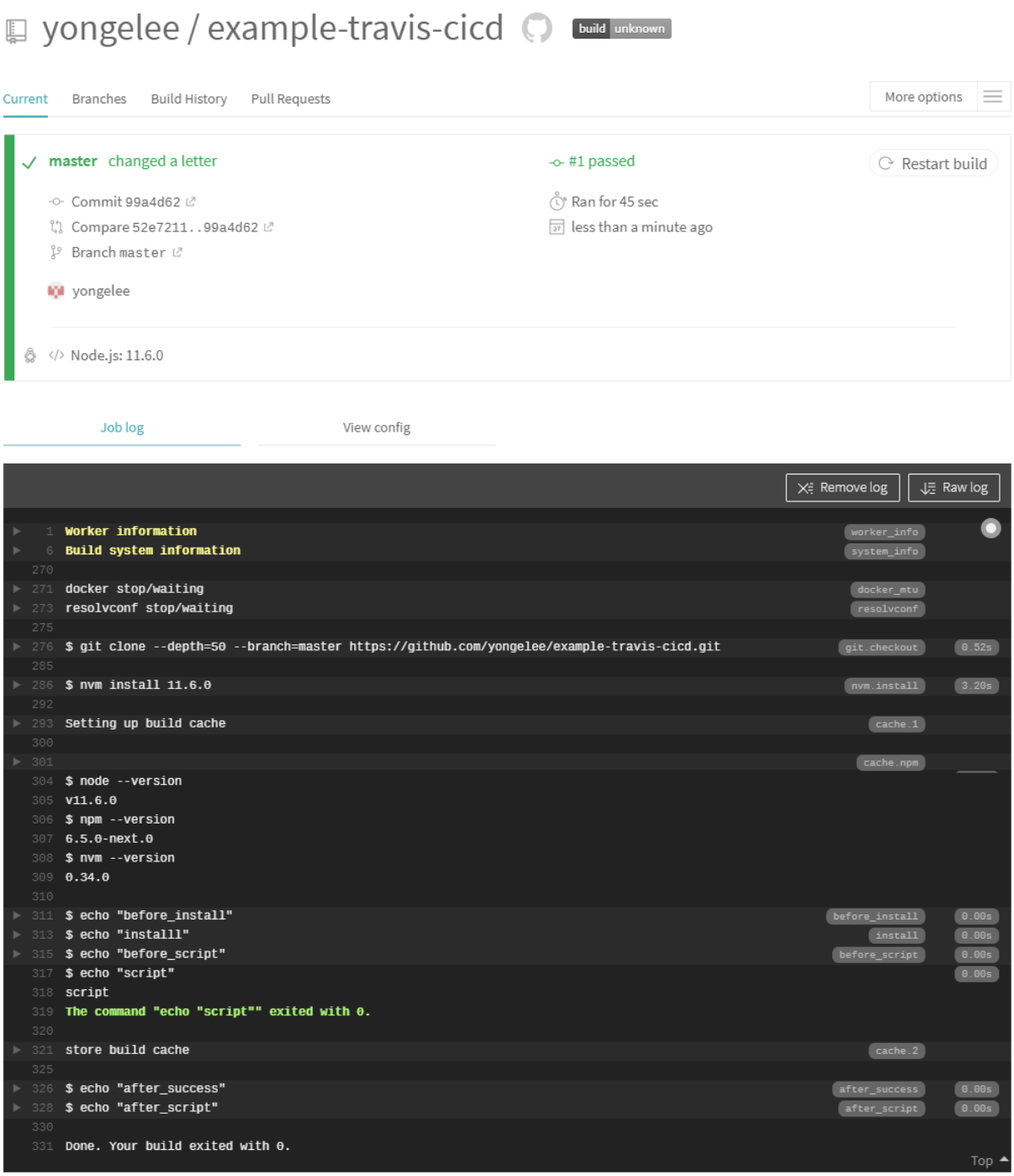
We added our .travis.yml file to the repository but it didn't trigger a build. Add a small change like adding a letter in one of your echo statements, then commit and push the changes.

After you've pushed your code, the interface in Travis should start to update after a few minutes. In the page for your Travis app, you should see a page that looks like this:



Yellow indicates that Travis has received the new commit with the .travis.yml file and is beginning to run the pipeline.

After the pipeline is finished, your page should look similar to this:

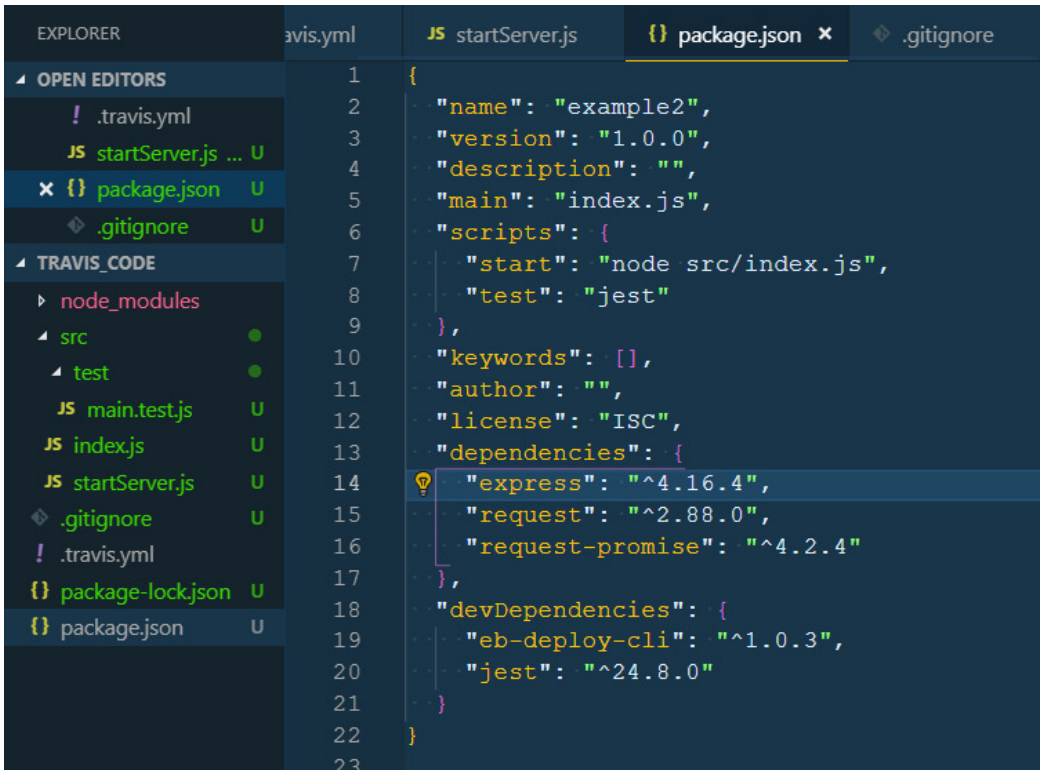


As you can see, the echo statements we've created are run in sequence. To view the full logs click "Raw log" in the top right corner.

Now that our pipeline is running in Travis, let's turn this into a pipeline that builds, tests, and deploys the repository.

Add the sample project

Our GitHub repository is empty other than the .travis.yml file. Let’s add the sample Node.js API project.Your project should look like this:



This is the same code base we’ve worked on before, the only difference is the .travis.yml file instead of the .gitlab-ci.yml file.

Add the build and test jobs configuration

To add build and test jobs to Travis, we need to add commands to the appropriate sections.

Since Travis has stages predefined, we have to use the ones they are expecting, in this case the stages we have to add commands to are: “install” and “script.”

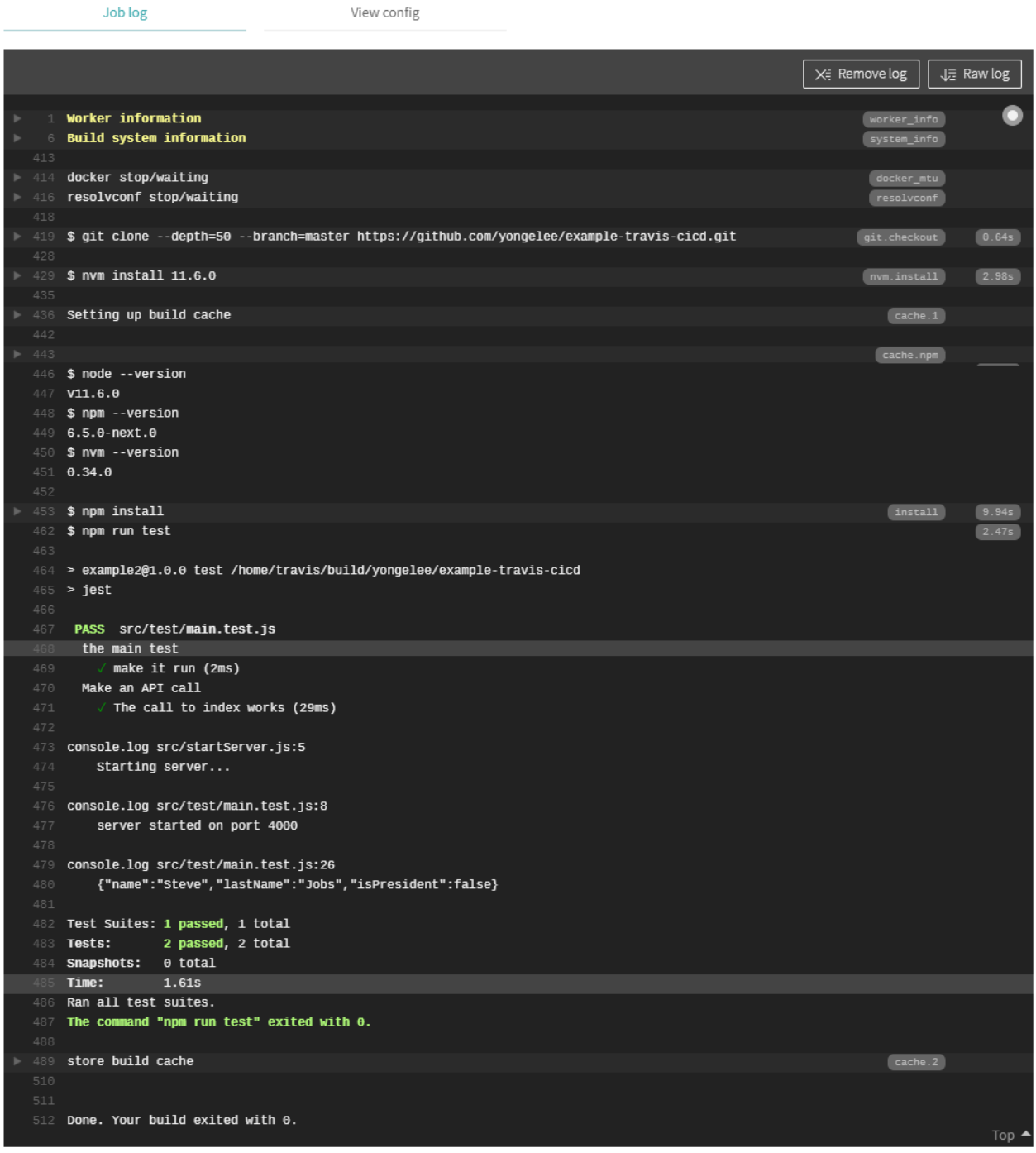
In install, that is the same thing as the build stage in GitLab. The script stage is the same as the test stage in GitLab.

So what do we have to do for our project? In install, we want to install the packages. The command would be “npm install.” In script we want to run our test so we run “npm run test.” Let’s add these commands to our yaml file.

Your .travis.yml file should now look like this gist.

Commit and push your changes.

Once your new pipeline is done running in Travis, your screen should look similar to this:



Add deploy configuration

Now that our install and test script works properly, we are ready to deploy our app to AWS. Thankfully, Travis CI has a deployment option available for AWS Elastic Beanstalk which makes it very easy!

For this to work we need our access key id, secure access key, and the AWS Elastic Beanstalk information like the app name, environment name, and s3 bucket name.

After you have all of that information, we need to add these values to our environment variables.

To add environment variables to Travis, go to your Travis pipeline page and go to “more options” and then “settings.”

You will then land on a page to fill out the environment variables. Add the name and value for your AWS settings.

Now we have all the information we need in our environment to deploy to AWS.

The deploy configuration for Travis is different from GitLab since we are using the integrated AWS Elastic Beanstalk option. In this link to the Travis CI documentation page for Elastic Beanstalk deployment, we can see that we just need to add the environment variable values to our configuration and Travis will automatically deploy to Elastic Beanstalk.

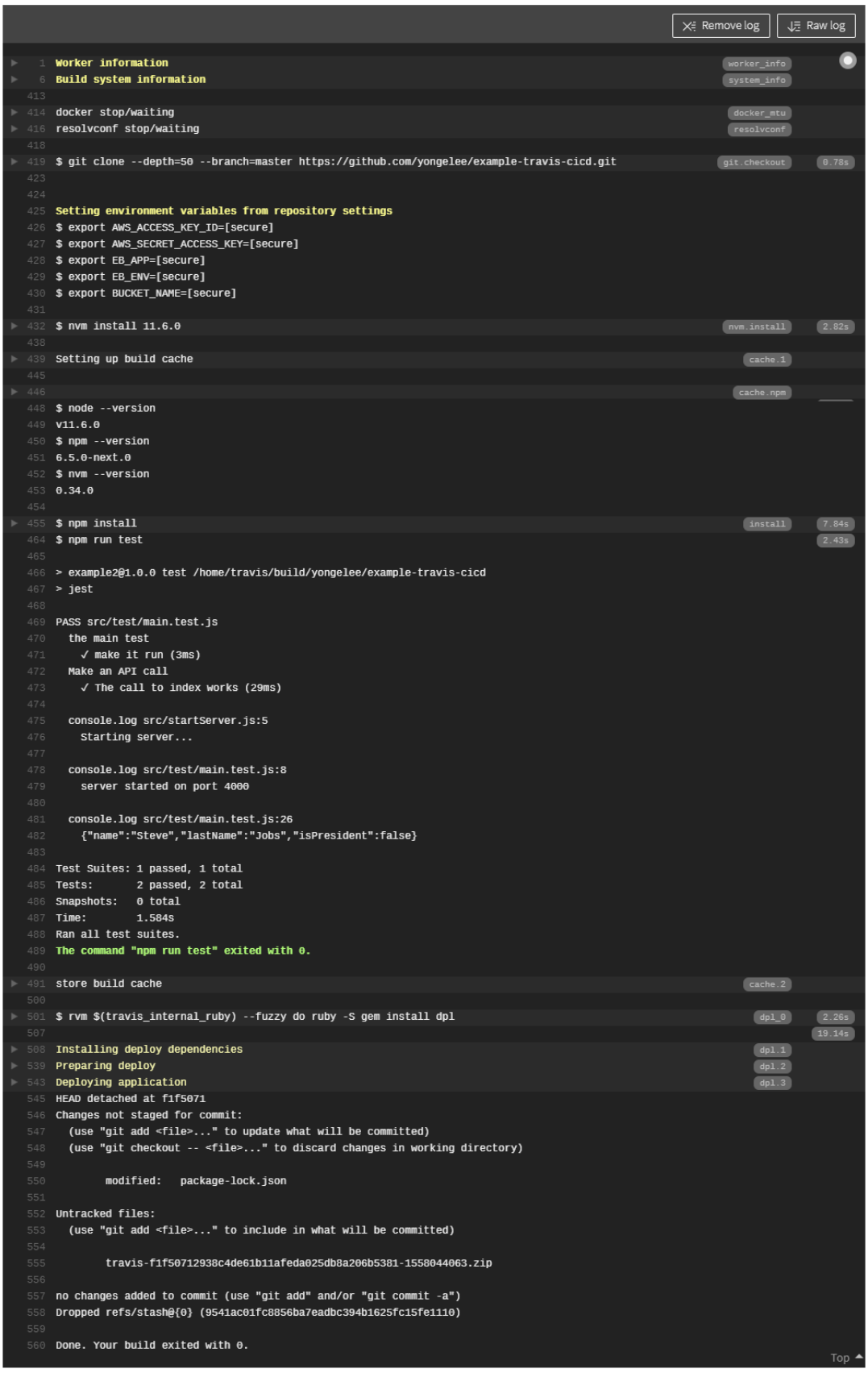
In the deploy section of the yaml file, we need to set the provider to elasticbeanstalk and fill in the blanks for access key, secret key, region, app, env, and bucket name.

Your .travis.yml file should now look like this gist.

To access your environment variables in the .travis.yml file you just put a "$" in front of the environment variable name.

And with that, our travis pipeline is complete!

Let's commit and push the changes and see what happens.


Job log
View config
Remove log
Raw log
1 Worker information worker_info
6 Build system information system_info
413
414 docker stop/waiting docker_mtu
416 resolvconf stop/waiting resolvconf
418
419 $ git clone --depth=50 --branch=master https://github.com/yongelee/example-travis-cicd.git git.checkout 0.78s
423
424
425 Setting environment variables from repository settings
426 $ export AWS_ACCESS_KEY_ID=[secure]
427 $ export AWS_SECRET_ACCESS_KEY=[secure]
428 $ export EB_APP=[secure]
429 $ export EB_ENV=[secure]
430 $ export BUCKET_NAME=[secure]
431
432 $ nvm install 11.6.0 nvm.install 2.82s
438
439 Setting up build cache cache.1
445
446 cache.npm
448 $ node --version
449 v11.6.0
450 $ npm --version
451 6.5.0-next.0
452 $ nvm --version
453 0.34.0
454
455 $ npm install install 7.84s
464 $ npm run test 2.43s
465
466 > example2@1.0.0 test /home/travis/build/yongelee/example-travis-cicd
467 > jest
468
469 PASS src/test/main.test.js
470 the main test
471 ✓ make it run (3ms)
472 Make an API call
473 ✓ The call to index works (29ms)
474
475 console.log src/startServer.js:5
476 Starting server...
477
478 console.log src/test/main.test.js:8
479 server started on port 4000
480
481 console.log src/test/main.test.js:26
482 {"name":"Steve","lastName":"Jobs","isPresident":false}
483
484 Test Suites: 1 passed, 1 total
485 Tests: 2 passed, 2 total
486 Snapshots: 0 total
487 Time: 1.584s
488 Ran all test suites.
489 The command "npm run test" exited with 0.
490
491 store build cache cache.2
500
501 $ rvm $(travis_internal_ruby) --fuzzy do ruby -S gem install dpl dpl_0 2.26s
507 19.14s
508 Installing deploy dependencies dpl.1
539 Preparing deploy dpl.2
543 Deploying application dpl.3
545 HEAD detached at f1f5071
546 Changes not staged for commit:
547 (use "git add <file>..." to update what will be committed)
548 (use "git checkout -- <file>..." to discard changes in working directory)
549
550 modified: package-lock.json
551
552 Untracked files:
553 (use "git add <file>..." to include in what will be committed)
554
555 travis-f1f50712938c4de61b11afeda025db8a206b5381-1558044063.zip
556
557 no changes added to commit (use "git add" and/or "git commit -a")
558 Dropped refs/stash@{0} (9541ac01fc8856ba7eadbc394b1625fc15fe1110)
559
560 Done. Your build exited with 0.
Top

If you see something like the console above, congratulations, you've just created a build, test, and deploy pipeline with GitHub and Travis CI!

You can check if your deployment was successful by going to your Elastic Beanstalk page. If there is a green check mark that means it was successful! Check by clicking the url for your environment.

If you've followed along with the sample project, your screen should look like this:

{"name":"Stevez","lastName":"Jobs","isPresident":false}

Try making some changes and commits to your code to see how your pipeline makes it so easy to deploy to production!

Review of CI/CD with GitHab and Travis

The steps we took to create a pipeline with GitHub, Travis CI, and Elastic Beanstalk were:

1. Add a .travis-ci.ymi file to GitHub repository
2. Sign up/in to travis-ci.org
3. Create CI/CD configuration file for message printing
4. Add the sample project
5. Add the build and test jobs configuration
6. Add the deploy configuration

There are a lot more configuration options available for many stages in this CI/CD pipeline. The pipeline we created is just a basic one!

A step closer to bug-free development

If you made it through those tutorials, congratulations! You've added a great tool to your arsenal as a developer or devops engineer.

Refactoring code you wrote days, weeks, months, or even years ago is a job very few people enjoy doing. To avoid this headache of a task, write some tests and add them to your pipeline early in your development process. That way your code refactoring can be done while the code you wrote is still fresh in your mind.

Do you know who else is your great bug-tracking buddy? Your users. All users from your department and the company, from beta users and actual customers, should have easily accessible channels to report issues or share their feedback.

Use a light-weight feedback tool to empower all users to submit issues as they experience it. This will bring you

closer to maintaining a bug-free product.

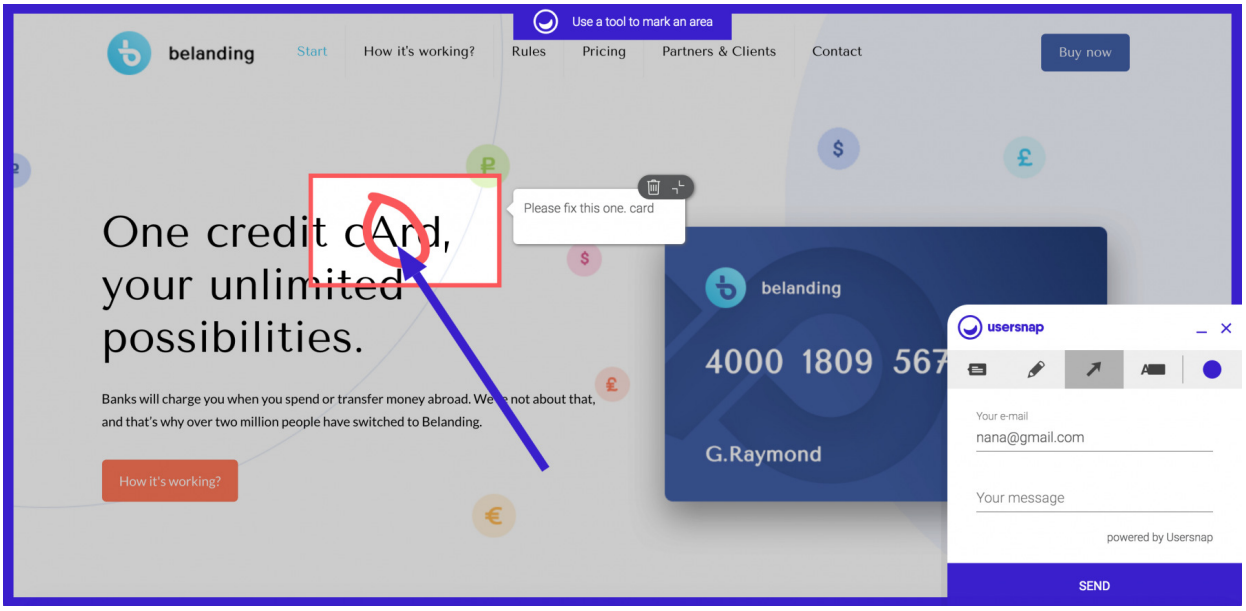
The key advantage of Usersnap's visual feedback feature is to capture and show the bug within the user's browser and automatically attach metadata such as URL, screen size, and console log to each ticket. The developers can resolve matters faster and more accurately, which in result will speed up delivery turnover.

Your future self and your future team will thank you for putting in the upfront work of adding tests and user feedback tools into your project from the start.

Like it. Share it.

Integrate Usersnap with GitLab and GitHub

Usersnap is a visual feedback tool for you to get instant in-product feedback from testers and users. The main idea is to communicate via screenshots, which can be annotated with click-and-drag arrows, virtual sticky notes, and highlighters directly in your browser.



You immediately see on the screen what the issue or suggestion is. Instead of sending multiple emails, creating confusion and wasting time, the feedback gets

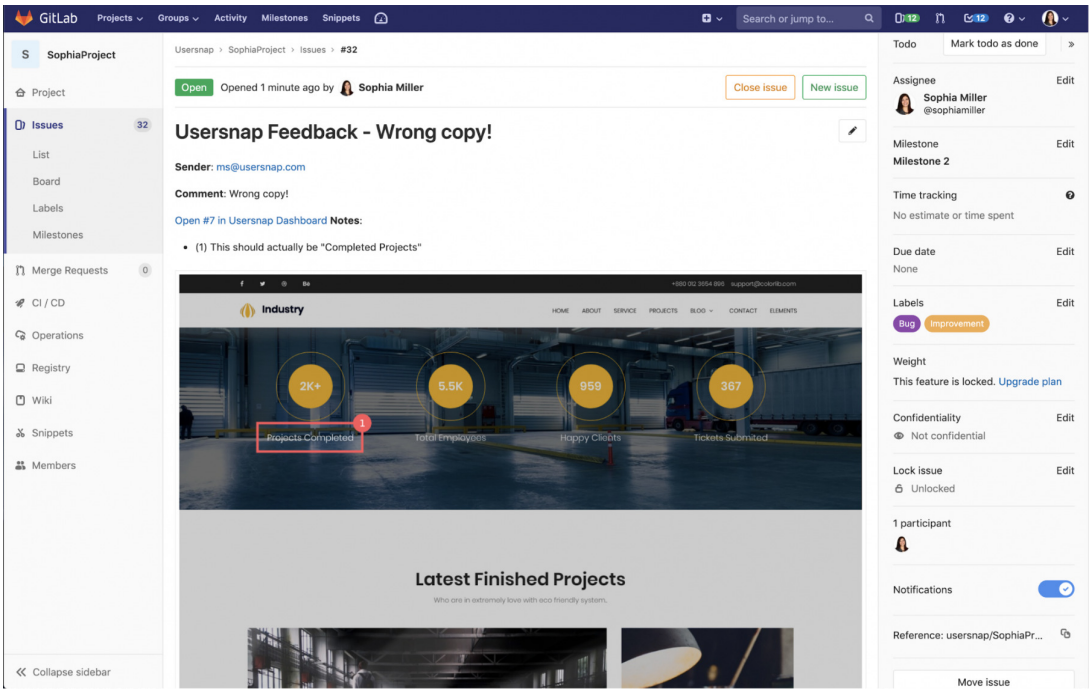
usersnap

shown instead of explained. The problem gets solved.

> I never want to go back to not using a bug tracker on staging sites. Usersnap seriously pays for itself in time and energy saved.
>
> - Nora Lahl, Director of Client Engagement at Lightburn Web Agency

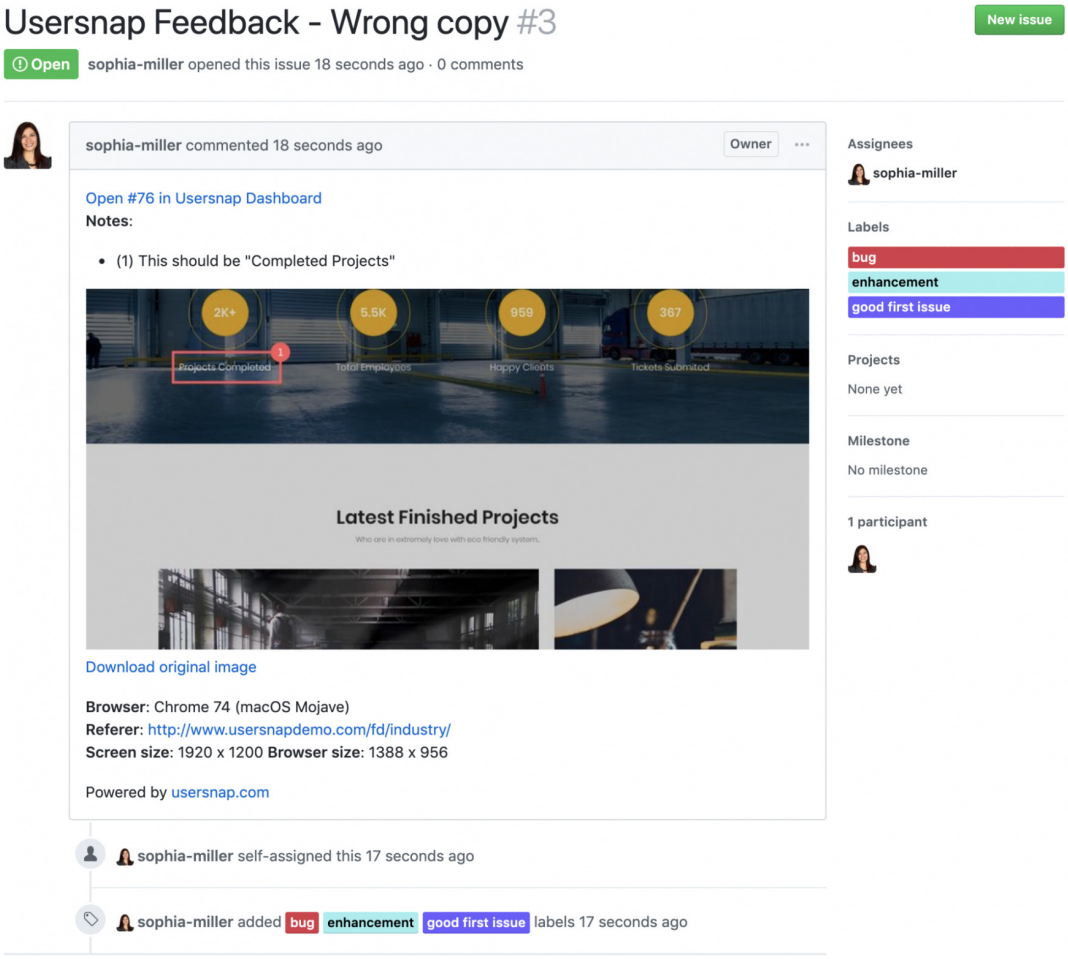
Once you connect Usersnap with GitLab or GitHub, an annotated screenshot with important browser information is sent to your project every time feedback is created with Usersnap.

This is a bug ticket created by Usersnap sent to GitLab:



usersnap

And this is how it looks like with GitHub:



> It has greatly facilitated our testing process. I love how it automatically captures the user's URL, screen resolution, OS, and browser data.
>
> - Rachel Panush, VP of Operations at Executionists

usersnap

Bring developers, designers and project managers on the same page. Fix and reproduce bugs faster with Usersnap.

All you need to do is install Usersnap on your website or application or use our browser extensions to receive bug reports and feedback.

Want to enhance your CI/CD development with more user feedback today?

Try Usersnap for free now >
Learn more about Usersnap and 20+ integrations >

> The in-app annotation feedback via Usersnap is accurate and fast. It helped us shorten the customer support cycle and thus plays an important role in maintaining and improving the usability of our product.
>
> - Joscha Feth, Engineer at Canva