

How To Build Docker Images and Host a Docker Image Repository with GitLab



Posted February 12, 2018  70.7k DOCKER GIT CI/CD UBUNTU 16.04

By: Brian Boucheron

Introduction

Containerization is quickly becoming the most accepted method of packaging and deploying applications in cloud environments. The standardization it provides, along with its resource efficiency (when compared to full virtual machines) and flexibility, make it a great enabler of the modern *DevOps* mindset. Many interesting *cloud native* deployment, orchestration, and monitoring strategies become possible when your applications and microservices are fully containerized.

Docker containers are by far the most common container type today. Though public Docker image repositories like Docker Hub are full of containerized open source software images that you can `docker pull` and use today, for private code you'll need to either pay a service to build and store your images, or run your own software to do so.

GitLab Community Edition is a self-hosted software suite that provides Git repository hosting, project tracking, CI/CD services, and a Docker image registry, among other features. In this tutorial we will use GitLab's continuous integration service to build Docker images from an example Node.js app. These images will then be tested and uploaded to our own private Docker registry.

Prerequisites

Before we begin, we need to set up a **secure GitLab server**, and a **GitLab CI runner** to execute continuous integration tasks. The sections below will provide links and more details.

A GitLab Server Secured with SSL

To store our source code, run CI/CD tasks, and host the Docker registry, we need a GitLab instance installed on an Ubuntu 16.04 server. GitLab currently recommends a **server with at least 2 CPU cores and 4GB of RAM**. Additionally, we'll secure the server with SSL certificates from Let's Encrypt. To do so, you'll need a domain name pointed at the server.

You can complete these prerequisite requirements with the following tutorials:

- [How To Set Up a Host Name with DigitalOcean](#) will show you how to manage a domain with the DigitalOcean control panel
- [Initial Server Setup with Ubuntu 16.04](#) will get a non-root, sudo-enabled user set up, and enable Ubuntu's `ufw` firewall
- [How To Install and Configure GitLab on Ubuntu 16.04](#) will show you how to install GitLab and configure it with a free TLS/SSL certificate from Let's Encrypt

A GitLab CI Runner

[How To Set Up Continuous Integration Pipelines with GitLab CI on Ubuntu 16.04](#) will give you an overview of GitLab's CI service, and show you how to set up a CI runner to process jobs. We will build on top of the demo app and runner infrastructure created in this tutorial.

Step 1 — Setting Up a Privileged GitLab CI Runner

In the prerequisite GitLab continuous integration tutorial, we set up a GitLab runner using `sudo gitlab-runner register` and its interactive configuration process. This runner is capable of running builds and tests of software inside of isolated Docker containers.

However, in order to build Docker images, our runner needs full access to a Docker service itself. The recommended way to configure this is to use Docker's official `docker-in-docker` image to run the jobs. This requires granting the runner a special `privileged` execution mode, so we'll create a second runner with this mode enabled.

Note: Granting the runner `privileged` mode basically disables all of the security advantages of using containers. Unfortunately, the other methods of enabling Docker-capable runners also carry similar security implications. Please look at [the official GitLab documentation on Docker Build](#) to learn more about the different runner options and which is best for your situation.

Because there are security implications to using a privileged runner, we are going to create a project-specific runner that will only accept Docker jobs on our `hello_hapi` project (GitLab admins can always manually add this runner to other projects at a later time). From your `hello_hapi` project page, click **Settings** at the bottom of the left-hand menu, then click **CI/CD** in the submenu:

Wiki

Snippets

Settings

General

Members

Integrations

Repository

CI / CD

Variables are applied to environments via the runner. They can be protected by only exposing them to protected environments. You can use variables for passwords, secret keys, or whatever you want.

Pipeline triggers

Triggers can force a specific branch or tag to get rebuilt with an API call. These tokens will impersonate their creator, including their access to projects and their project permissions.

Now click the **Expand** button next to the **Runners settings** section:

Runners settings

Register and see your runners for this project.

Expand

There will be some information about setting up a **Specific Runner**, including a registration token. Take note of this token. When we use it to register a new runner, the runner will be locked to this project only.

Specific Runners

How to setup a specific Runner for a new project

1. Install a Runner compatible with GitLab CI (checkout the [GitLab Runner](#) section for information on how to install it).
2. Specify the following URL during the Runner setup: `https://gitlab.example.com/`
3. Use the following registration token during setup: `Dr9uTy53dVgwZ4bs7BLN`
4. Start the Runner!

Shared Runners

GitLab Shared Runners execute code of different projects on the same Runner unless you configure GitLab Runner Autoscale with MaxBuilds 1 (which it is on GitLab.com).

Disable shared Runners

for this project

Available shared Runners : 1

● 374fac64

example-runner

#1

While we're on this page, click the **Disable shared Runners** button. We want to make sure our Docker jobs always run on our privileged runner. If a non-privileged shared runner was available, GitLab might choose to use that one, which would result in build errors.

Log in to the server that has your current CI runner on it. If you don't have a machine set up with runners already, go back and complete the [Installing the GitLab CI Runner Service](#) section of the prerequisite tutorial before proceeding.

Now, run the following command to set up the privileged project-specific runner:

```
$ sudo gitlab-runner register -n \
```

```
$ --url https://gitlab.example.com/ \
$ --registration-token your-token \
$ --executor docker \
$ --description "docker-builder" \
$ --docker-image "docker:latest" \
$ --docker-privileged
```

Output

```
Registering runner... succeeded runner=61SR6BwV
Runner registered successfully. Feel free to start it, but if it's running already the config should
```

Be sure to substitute your own information. We set all of our runner options on the command line instead of using the interactive prompts, because the prompts don't allow us to specify `--docker-privileged` mode.

Your runner is now set up, registered, and running. To verify, switch back to your browser. Click the wrench icon in the main GitLab menu bar, then click **Runners** in the left-hand menu. Your runners will be listed:

Type	Runner token	Description	Version	Projects	Jobs	Tags	Last contact	
<div>specific</div> <div>locked</div>	45eeb508	docker-builder	10.4.0	1	0		less than a minute ago	<div><div></div><div></div><div></div></div>
<div>shared</div>	374fac64	example-runner	10.4.0	n/a	0		21 minutes ago	<div><div></div><div></div><div></div></div>

Now that we have a runner capable of building Docker images, let's set up a private Docker registry for it to push images to.

Step 2 — Setting Up GitLab's Docker Registry

Setting up your own Docker registry lets you push and pull images from your own private server, increasing security and reducing the dependencies your workflow has on outside services.

GitLab will set up a private Docker registry with just a few configuration updates. First we'll set up the URL where the registry will reside. Then we will (optionally) configure the registry to use an S3-compatible object storage service to store its data.

SSH into your GitLab server, then open up the GitLab configuration file:

```
$ sudo nano /etc/gitlab/gitlab.rb
```

Scroll down to the **Container Registry settings** section. We're going to uncomment the `registry_external_url` line and set it to our GitLab hostname with a port number of 5555 :

```
registry_external_url 'https://gitlab.example.com:5555'
```

Next, add the following two lines to tell the registry where to find our Let's Encrypt certificates:

```
                                /etc/gitlab/gitlab.rb

registry_nginx['ssl_certificate'] = "/etc/letsencrypt/live/gitlab.example.com/fullchain.pem"
registry_nginx['ssl_certificate_key'] = "/etc/letsencrypt/live/gitlab.example.com/privkey.pem"
```

Save and close the file, then reconfigure GitLab:

```
$ sudo gitlab-ctl reconfigure
```

Output

```
. . .
gitlab Reconfigured!
```

Update the firewall to allow traffic to the registry port:

```
$ sudo ufw allow 5555
```

Now switch to another machine with Docker installed, and log in to the private Docker registry. If you don't have Docker on your local development computer, you can use whichever server is set up to run your GitLab CI jobs, as it has Docker installed already:

```
$ docker login gitlab.example.com:5555
```

You will be prompted for your username and password. Use your GitLab credentials to log in.

Output

```
Login Succeeded
```

Success! The registry is set up and working. Currently it will store files on the GitLab server's local filesystem. If you'd like to use an object storage service instead, continue with this section. If not, skip down to Step 3.

To set up an object storage backend for the registry, we need to know the following information about our object storage service:

- **Access Key**
- **Secret Key**

- **Region** (us-east-1) for example, if using Amazon S3, or **Region Endpoint** if using an S3-compatible service (<https://nyc.digitaloceanspaces.com>)
- **Bucket Name**

If you're using DigitalOcean Spaces, you can find out how to set up a new Space and get the above information by reading [How To Create a DigitalOcean Space and API Key](#).

When you have your object storage information, open the GitLab configuration file:

```
$ sudo nano /etc/gitlab/gitlab.rb
```

Once again, scroll down to the container registry section. Look for the `registry['storage']` block, uncomment it, and update it to the following, again making sure to substitute your own information where appropriate:

```
                                /etc/gitlab/gitlab.rb

registry['storage'] = {
  's3' => {
    'accesskey' => 'your-key',
    'secretkey' => 'your-secret',
    'bucket' => 'your-bucket-name',
    'region' => 'nyc3',
    'regionendpoint' => 'https://nyc3.digitaloceanspaces.com'
  }
}
```

If you're using Amazon S3, you only need `region` and not `regionendpoint`. If you're using an S3-compatible service like Spaces, you'll need `regionendpoint`. In this case `region` doesn't actually configure anything and the value you enter doesn't matter, but it still needs to be present and not blank.

Save and close the file.

Note: There is currently a bug where the registry will shut down after thirty seconds if your object storage bucket is empty. To avoid this, put a file in your bucket before running the next step. You can remove it later, after the registry has added its own objects.

If you are using DigitalOcean Spaces, you can drag and drop to upload a file using the Control Panel interface.

Reconfigure GitLab one more time:

```
$ sudo gitlab-ctl reconfigure
```

On your other Docker machine, log in to the registry again to make sure all is well:

```
$ docker login gitlab.example.com:5555
```

You should get a `Login Succeeded` message.

Now that we've got our Docker registry set up, let's update our application's CI configuration to build and test our app, and push Docker images to our private registry.

Step 3 — Updating `gitlab-ci.yml` and Building a Docker Image

Note: If you didn't complete the [prerequisite article on GitLab CI](#) you'll need to copy over the example repository to your GitLab server. Follow the [Copying the Example Repository From GitHub](#) section to do so.

To get our app building in Docker, we need to update the `.gitlab-ci.yml` file. You can edit this file right in GitLab by clicking on it from the main project page, then clicking the **Edit** button. Alternately, you could clone the repo to your local machine, edit the file, then `git push` it back to GitLab. That would look like this:

```
$ git clone git@gitlab.example.com:sammy/hello_hapi.git
$ cd hello_hapi
$ # edit the file w/ your favorite editor
$ git commit -am "updating ci configuration"
$ git push
```

First, delete everything in the file, then paste in the following configuration:

```
.gitlab-ci.yml

image: docker:latest
services:
- docker:dind

stages:
- build
- test
- release

variables:
  TEST_IMAGE: gitlab.example.com:5555/sammy/hello_hapi:$CI_COMMIT_REF_NAME
  RELEASE_IMAGE: gitlab.example.com:5555/sammy/hello_hapi:latest

before_script:
- docker login -u gitlab-ci-token -p $CI_JOB_TOKEN gitlab.example.com:5555
```

```
build:
  stage: build
  script:
    - docker build --pull -t $TEST_IMAGE .
    - docker push $TEST_IMAGE

test:
  stage: test
  script:
    - docker pull $TEST_IMAGE
    - docker run $TEST_IMAGE npm test

release:
  stage: release
  script:
    - docker pull $TEST_IMAGE
    - docker tag $TEST_IMAGE $RELEASE_IMAGE
    - docker push $RELEASE_IMAGE
  only:
    - master
```

Be sure to update the highlighted URLs and usernames with your own information, then save with the **Commit changes** button in GitLab. If you're updating the file outside of GitLab, commit the changes and `git push` back to GitLab.

This new config file tells GitLab to use the latest docker image (`image: docker:latest`) and link it to the docker-in-docker service (`docker:dind`). It then defines `build`, `test`, and `release` stages. The `build` stage builds the Docker image using the `Dockerfile` provided in the repo, then uploads it to our Docker image registry. If that succeeds, the `test` stage will download the image we just built and run the `npm test` command inside it. If the test stage is successful, the `release` stage will pull the image, tag it as `hello_hapi:latest` and push it back to the registry.

Depending on your workflow, you could also add additional `test` stages, or even `deploy` stages that push the app to a staging or production environment.

Updating the configuration file should have triggered a new build. Return to the `hello_hapi` project in GitLab and click on the CI status indicator for the commit:



Update .gitlab-ci.yml

Administrator authored less than a minute ago



0c5c5fef



On the resulting page you can then click on any of the stages to see their progress:

Changes 1

Pipelines 1

Status	Pipeline	Commit	Stages
	#36 by latest	0c5c5fef Update .gitlab-ci.yml	

Administrator > hello_hapi > Jobs > #42

passed

Job #42 triggered 15 minutes ago by Administrator

```
Running with gitlab-runner 10.4.0 (857480b6)
on docker-builder (45eeb508)
Using Docker executor with image docker:latest ...
Starting service docker:dind ...
Pulling docker image docker:dind ...
Using docker image docker:dind ID=sha256:4a2625517421ec21f13a0c31683ab1cc6ac259974be47ba7f2f86cb0c7a9a714 for docker service...
Waiting for services to be up and running...
Using docker image sha256:ba8eb750d7526248d538e7f8bdb1f2115e9ea572c10df99db6685876f871bf4d for predefined container...
Pulling docker image docker:latest ...
Using docker image docker:latest ID=sha256:73a5b9800caa376abdb533a45a66f418d37443a3c7a9d475838eb5fe0ba0bd06 for build container...
Running on runner-45eeb508-project-2-concurrent-0 via gitlab...
Cloning repository...
Cloning into '/builds/sammy/hello_hapi'...
Checking out 0c5c5fef as master...
Skipping Git submodules setup
$ docker login -u gitlab-ci-token -p $CI_JOB_TOKEN gitlab.boink.pro:5555
WARNING! Using --password via the CLI is insecure. Use --password-stdin
```

build

Duration: 1 minute 41 seconds

Runner: #2

Commit 0c5c5fef

Update .gitlab-ci.yml

Pipeline #36 from master

build

build

Eventually, all stages should indicate they were successful by showing green check mark icons. We can find the Docker images that were just built by clicking the **Registry** item in the left-hand menu:

Container Registry

With the Docker Container Registry integrated into GitLab, every project can have its own space to store its Docker images.

Learn more about Container Registry.

sammy/hello_hapi

Tag	Tag ID	Size	Created
latest	4f036568d	266.48 MiB	3 minutes ago
master	4f036568d	266.48 MiB	3 minutes ago

If you click the little "document" icon next to the image name, it will copy the appropriate `docker pull ...` command to your clipboard. You can then pull and run your image:

```
$ docker pull gitlab.example.com:5555/sammy/hello_hapi:latest
$ docker run -it --rm -p 3000:3000 gitlab.example.com:5555/sammy/hello_hapi:latest
```

Output

```
> hello@1.0.0 start /usr/src/app
> node app.js
```

Server running at: <http://56fd5df5ddd3:3000>

The image has been pulled down from the registry and started in a container. Switch to your browser and connect to the app on port 3000 to test. In this case we're running the container on our local machine, so we can access it via **localhost** at the following URL:

`http://localhost:3000/hello/test`

Output

Hello, test!


Success! You can stop the container with CTRL-C . From now on, every time we push new code to the master branch of our repository, we'll automatically build and test a new `hello_hapi:latest` image.

Conclusion

In this tutorial we set up a new GitLab runner to build Docker images, created a private Docker registry to store them in, and updated a Node.js app to be built and tested inside of Docker containers.

To learn more about the various components used in this setup, you can read the official documentation of [GitLab CE](#), [GitLab Container Registry](#), and [Docker](#).

By: Brian Boucheron

 Upvote (18)  Subscribe  Share



We just made it easier for you to deploy faster.

[TRY FREE](#)

Related Tutorials

How to Manually Set Up a Prisma Server on Ubuntu 18.04

How To Use Traefik as a Reverse Proxy for Docker Containers on Debian 9

How To Set Up a Private Docker Registry on Ubuntu 18.04

How To Secure a Containerized Node.js Application with Nginx, Let's Encrypt, and Docker Compose

How To Set Up Multi-Node Deployments With Rancher 2.1, Kubernetes, and Docker Machine on Ubu...

5 Comments

Leave a comment...

Log In to Comment

 [stoivo](#) February 28, 2018

 1 In your .gitlabci.yml you have

```
test:
  stage: test
  script:
    - docker pull $TEST_IMAGE
    - docker run $TEST_IMAGE npm test
```

Could you do

```
test:
  stage: test
  image: $TEST_IMAGE
  script:
    - npm test
```

I think that look a lot cleaner. But I don't know if it works.

And with this setup, you would get a shit load of images on the Container register. I will not use the images produced from before it is merged into master. So I was thinking that I when the job succeeds or fails I would like to destroy the image again.

Do you know how we can set up that?

 [maybreezes](#) *March 7, 2018*

o I agree with you. Looking for solutions!

 [njam](#) *May 25, 2018*

o In the `.gitlabci.yml` 's *build* step you have:

```
- docker build --pull -t $TEST_IMAGE .
```

I think this will never pull any layers because `$TEST_IMAGE` doesn't exist yet, and thus not reuse/cache anything?

How about pulling previous layers from `$RELEASE_IMAGE` ?

```
- docker pull $RELEASE_IMAGE
- docker build -t $TEST_IMAGE .
```

 [njam](#) *May 27, 2018*

o Actually it needs to be like this to use caching:

```
- docker pull $RELEASE_IMAGE
- docker build --pull --cache-from $RELEASE_IMAGE -t $TEST_IMAGE .
```

See the comment and discussion here: https://gitlab.com/gitlab-org/gitlab-ce/issues/17861#note_19140733

 [piotrjankiewicz02](#) *June 13, 2018*

o Hi everyone!

If you have a problem with logging in after registering new runner on port 5555 with those settings:

```
registry_nginx['ssl_certificate'] = "/etc/letsencrypt/live/gitlab.example.com/fullchain.pem"
registry_nginx['ssl_certificate_key'] = "/etc/letsencrypt/live/gitlab.example.com/privkey.pem"
```

Try those instead:

```
nginx['ssl_certificate'] = "/etc/gitlab/ssl/#{node['fqdn']}.crt"  
nginx['ssl_certificate_key'] = "/etc/gitlab/ssl/#{node['fqdn']}.key"
```

It will help you avoid this ugly error:

```
docker login Error response from daemon connection refused
```

When executing this:

```
docker login gitlab.example.com:5555
```

Hope it helped. Cheers!



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.



Copyright © 2019 DigitalOcean™ Inc.

[Community](#) [Tutorials](#) [Questions](#) [Projects](#) [Tags](#) [Newsletter](#) [RSS](#) 

[Distros & One-Click Apps](#) [Terms, Privacy, & Copyright](#) [Security](#) [Report a Bug](#) [Write for DOnations](#) [Shop](#)

