IaC CI with GitLab and Terraform

Marko Skender
Oct 1, 2020 · 5 min read

There are lots of options nowadays to describe and actually deploy your infrastructure as a code (IaC) — here we are gonna take a look at how to set up some quick& dirty continuous integration (CI) using the increasingly popular Terraform tool to spin up IaC and Gitlab as a source-control and CI platform.

How does it work? Well, ideally, you simply edit your Terraform resource files using your favorite editor (MS Word not allowed), push the repo via git to Gitlab and the GitLab's CI Pipeline will make required changes directly to your infrastructure. There's a couple components to this:

- your local git and Terraform tools for managing the repo and the code
- Gitlab instance for source and CI job control/triggers
- A machine running Gitlab CI agent(Gitlab runner) to execute jobs
- Terraform environment on Gitlab runner machine to CI-execute terraform plans.

First of all, you should start by setting up your Terraform repo on your Gitlab instance, and clone it locally (yes, of course, you should have git!). We'll presuppose your Gitlab's URL as `gitlab.com` for example sake.

You should also download and install the Terraform binary from https://www.terraform.io/.

Create a file `backend.tf` in project's root so Terraform knows to use HTTP state backend:

```
terraform {
 backend "http" {}
```

```
}
```

After that's said and done, you should initialize the Terraform workspace (with a bit of a twist):

```
terraform init \
-backend-
config="address=https://gitlab.com/api/v4/projects/<PROJECT_ID>/terr
aform/state/<YOUR_STATE>" \
-backend-
config="lock_address=https://gitlab.com/api/v4/projects/<PROJECT_ID>
/terraform/state/<YOUR_STATE>/lock" \
-backend-
config="unlock_address=https://gitlab.com/api/v4/projects/<PROJECT_I
D>/terraform/state/<YOUR_STATE>/lock" \
-backend-config="username=<YOUR_USERNAME>" \
-backend-config="password=<YOUR_TOKEN>" \
-backend-config="lock_method=POST" \
-backend-config="unlock_method=DELETE" \
-backend-config="retry_wait_min=5"
```

Terraform backend is usually stored locally, but we declare it as a Http remote in `backend.tf` and initialize it on your Gitlab's terraform project so it remains consistent between various agents using it (including the Gitlab runner). Otherwise each agent (human or machine) will have its own state which leads to creating duplicates of defined infrastructure — and on each run of the CI pipeline!

Replace <PROJECT_ID> with your project's repository ID (it's beneath the project name in Gitlabs Web UI), <YOUR_STATE> with whatever label you desire (I'm using "production" as per Gitlab docs), <YOUR_USERNAME> with, well, your Gitlab's username and also generate yourself an API token and replace <YOUR_TOKEN> with it.

We're now prepared to install a Gitlab runner for the job somewhere. We have a couple of options here:

- install it on a machine and run Terraform docker image to execute jobs
- install it on a Kubernetes cluster and run Terraform pod/docker image to execute jobs
- install it on a machine and run Terraform + wrapper script to execute jobs

The first two cases are documented on https://docs.gitlab.com/ee/user/infrastructure/. I have opted for wrapper script on the Gitlab instance itself since I'm running LXC and can't run containers inside containers. Not only that, but I needed a custom provider module for Proxmox, so that would only complicate stuff (I'd also need to rebuild the docker image).

This is your basic .gitlab-runner.yml (if you use Docker, just uncomment the image definition):

```
default:
 ## image: registry.gitlab.com/gitlab-org/terraform-
images/stable:latest

variables:
  TF_ROOT: ${CI_PROJECT_DIR}
  TF_ADDRESS:
${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/terraform/state/<YOUR
STATE>
  TF_PASSWORD: <YOUR_TOKEN>

cache:
  key: example-production
  paths:
    - ${TF_ROOT}/.terraform

before_script:
  - cd ${TF_ROOT}

stages:
  - prepare
  - validate
  - build
  - deploy

init:
  stage: prepare
  script:
    - gitlab-terraform init

validate:
  stage: validate
  script:
    - gitlab-terraform validate

plan:
  stage: build
  script:
    - gitlab-terraform plan

  artifacts:
    name: plan
    paths:
```

```
      - ${TF_ROOT}/plan.cache

apply:
  stage: deploy
  environment:
    name: production
  script:
    - gitlab-terraform apply
  dependencies:
    - plan
  #when: manual
  only:
    - master
```

The file needs be saved in the project's root exactly as `.gitlab-ci.yml`. Don't touch vars in curly braces, they will be expanded automatically - the only thing needs replacing is <YOUR STATE> and <YOUR_TOKEN> using the values you used in `terraform init`. You can also uncomment `when: manual` which then requires you to manually run apply stage through Gitlabs Web UI (open the project's page, it's under CI/CD -> Jobs).

Since we're not running a docker image, we'll need a machine with the following:

- GitLab-runner (regardless of using docker or no)
- terraform binary (same version as on your machine, v.0.12 or 0.13)
- GitLab-terraform wrapper script (referenced in gitlab-ci.yml)

GitLab-terraform you can download get here, rename, make executable, and put inside path, i.e:

```
wget https://gitlab.com/gitlab-org/terraform-images/-/raw/master/src/bin/gitlab-terraform.sh -O /bin/gitlab-terraform

chmod +x /bin/gitlab-terraform
```

The rest is well documented here:

> Install GitLab Runner
>
> GitLab Runner can be installed and used on GNU/Linux, macOS, FreeBSD, and Windows. There are three ways to install it...
>
> docs.gitlab.com

Install Terraform

To use Terraform you will need to install it. HashiCorp distributes Terraform as a binary package. You can also install...

learn.hashicorp.com

That's mostly it…or is it?

You see, there might be a case in which you need a Terraform plugin outside Hashicorp's registry, which I'm not sure how's handled inside the referenced Docker image (I presume one should rebuild the image to include custom modules), so using a script on a machine makes this much easier. You should install the plugins the same way as on any other machine, just be careful to install them under `/home/gitlab-runner/.terraform/` plugins directory since that's the account the runner is running on.

Also, there's been a pretty significant change in local plugins declaration and placement between 0.12 and 0.13 versions, so read up:

Upgrading to Terraform v0.13 - Terraform by HashiCorp

Terraform v0.13 is a major release and thus includes some changes that you'll need to consider when upgrading. This...

www.terraform.io

In the end, you should test with some disposable infrastructure to see how it holds up on multiple changes, commits, etc. Provisioning infrastructure is no joke (despite setting up tools to do so being a breeze) —if you f** it up, the pain can be real!

If this post was helpful, please click the clap 👏 button below a few times to show your support for the author! ⬇

Infrastructure As Code DevOps Terraform Gitlab Ci