GitHub & GitLab CI/CD

Spring 2021

Contents

1 Introduction

A proper CI/CD setup allows software development teams to iterate quickly by automating some of the code review process: checking for proper code formatting, running all tests, and ensuring it builds without errors (and, hopefully, without warnings). Once new code merges with `master`, it also allows for that code to be automatically deployed to a staging / QA environment for it to be further tested and then be deployed to production.

Previously a separate CI/CD solution such as Jenkins was often used, however more recently both GitHub and GitLab have integrated CI/CD directly into their products. This guide aims to give an overview of both GitHub and GitLab's CI/CD solutions and walk through setting up both with a sample project.

1.1 GitHub Actions

Relatively recently, in August 2019[1], GitHub introduced its own CI/CD solution as part of GitHub Actions. For private repositories on free accounts, GitHub provides 2,000 free Linux-machine minutes per month, (1,000 if used on Windows or 200 if used on macOS)[2]. These minute limits are for GitHub-hosted runners - one can self-host a runner to not be subjected to these limits. For example, for as low as $2.50/mo, one could get unlimited* CI/CD minutes per month with Vultr (*unlimited-ish; note that one month does not last forever, although in 2020 they certainly felt like they did[3]).

1.2 GitLab CI/CD

GitLab CI/CD has been around for a few years longer, having been introduced in 2015[4]. For private repositories, GitLab only provides 400 free minutes and does not appear to provide any shared runners with macOS. GitLab also allows one to self-host a runner which are not subjected to minute-limits, same as GitHub.

[1] https://github.blog/2019-08-08-github-actions-now-supports-ci-cd/

[2] https://docs.github.com/en/github/setting-up-and-managing-billing-and-payments-on-github/about-billing-for-github-actions

[3] https://www.cbc.ca/news/canada/british-columbia/coronavirus-update-bc-what-you-need-to-know-march-21-1.5505496

[4] https://about.gitlab.com/releases/2015/09/22/gitlab-8-0-released/

2 Setup

For this guide I will be using a simple React app. This will allow for multiple stages within the CI/CD pipeline: a "test" stage to check the code format using Prettier, a "build" stage to build the production version of the app and a "deploy" stage to deploy the new code. While this is simpler than most CI/CD setups, it'll allow us to focus on the CI/CD itself rather than configuring a complex project first. I'll also be using self-hosted runners so I can deploy. The provided runners can be used for building and testing but not for deployments.

2.1 Project Setup

To follow along, you'll need to create an account with both GitHub and GitLab (or use your existing accounts) and install NodeJS (we'll need this in order to use `npm`) and Git. You'll also need a machine to host the CI/CD runners - I'm using a DigitalOcean Droplet for this purpose but anything running Linux should do.

Once you have the necessary accounts and software:

1. Create a new repo in GitHub
2. Create a new repo in GitLab
3. Create a new directory and run `git init`
4. Within the directory just created, create a React project: `npx create-react-app example-app`
5. Install Prettier within the project: `cd example-app && npm install prettier`
6. Add the files to Git and then commit
7. Add both remotes, using the actual URLs given by both GitHub and GitLab:
   (a) `git remote add origin git@github.com ...`
   (b) `git remote set-url --add --push origin git@github.com ...`
   (c) `git remote set-url --add --push origin git@gitlab.com ...`
   (d) `git push -u origin master`
   (e) In the future, a single `git push` will push to both remotes
8. That's it for the project setup! You could further modify the React app at this point, although this guide will just deploy the default template for simplicity.

2.2 Install Docker on Deployment Machine

Docker will be used for deployments and also as an environment for the GitHub and GitLab runners to run the test and build stages in. Follow the official instructions to install both Docker Engine and Docker Compose on the machine you plan to use for the self-hosted runners.

2.3 GitHub Actions Setup

To install the self-hosted runner, open your repository within GitHub. Under Settings → Actions, scroll to the bottom and click "Add runner". We need to create two runners (one for testing and building and the other for deployments). Since we need two of them, we'll use a different folder name than the suggested `actions-runner`.

1. Create the first in a folder named `github-docker`. When you reach the configure step, name the runner `docker` and give it a label of `docker`. The default work folder can be used. Skip running it for now.

2. Create the second in a folder named `github-deploy`. When you reach the configure step, name the runner `deploy` and give it a label of `deploy`. The default work folder can be used. Skip running it for now.

Then install both runners as a service. Within each runner directory, install the service using `sudo ./svc.sh install` and then start it with `sudo ./svc.sh start`.

2.4 GitLab CI/CD Setup

To install the self-hosted runner, open your repository within GitLab. On the left, navigate to Settings → CI/CD. Expand the "Runners" section and then click on "Show Runner installation instructions". Follow the given instructions to create two runners:

1. The first should use the `docker` executor, be tagged with `docker` and use `node:14` as the default image.

2. The second should use the `shell` executor and be tagged with `shell`

We'll refer to these tags in our CI/CD setup later on. The GitLab Runner will also need access to Docker - run: `sudo usermod -aG docker gitlab-runner` to give it access.

3 Configuring the Pipelines

3.1 Getting the Project Deployment-Ready

Although the React project can be run locally using `npm start` for development, a file server is needed to deploy for production. The easiest way to set this up is using Docker and Caddy. In the main project directory, create a file named `Caddyfile` with the following contents:

```
:80
file_server
root * /var/www
```

This configuration serves static files on port 80 from the `/var/www` directory. For simplicity, we'll just use plain HTTP. Caddy does have built-in automatic HTTPS but we'd have to configure a domain for that.

We'll also need a `docker-compose.yml` file in the main project directory. It should look like this:

```yaml
version: "3.7"
services:
    caddy:
        image: caddy:2.3.0
        restart: always
        volumes:
            - ./Caddyfile:/etc/caddy/Caddyfile
            - ./example-app/build:/var/www
        ports:
            - ${PORT}:80
```

Note how we map the built version of the app to the `/var/www` directory that Caddy is serving files from within Docker. We also map port 80 inside the container to the port contained in the `PORT` environment variable - this is mostly done so that the GitHub and GitLab deployments can deploy on separate ports rather than interfering with each other. In a normal scenario with only one CI/CD setup, one could just map port `80:80` directly. I cannot imagine a sane setup in which it would make sense to actually use both GitHub and GitLab to manage deployments for the same project - both are used here for comparison purposes only.

3.2 GitHub Configuration

GitHub Actions are configured as a YAML configuration file within a `.github/workflows` folder inside repo itself (and thus the configuration is also version-controlled). GitHub has extensive documentation explaining all of the different configuration options. Rather than trying to explain my configuration in paragraph-form, here is a heavily commented version that will test, build and deploy the example project:

```yaml
# Specify a name for this workflow
name: CICD

# Run this workflow on pushes to master and pull requests into master
on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]

jobs:
  test-react:
    # Run this job only on runners with both of these labels
    runs-on: [self-hosted, docker]
    # Run using the `node:14` image in Docker
    container: node:14
    steps:
      # First, checkout the repo
      - uses: actions/checkout@v2
      # Second, run Prettier
      # Also name the step (will be shown in GitHub's UI; optional)
      - name: Run Prettier
        run: |
          cd example-app
          npm install
          npx prettier --check .

  build-react:
    needs: test-react
    runs-on: [self-hosted, docker]
    container: node:14
    # Specify environment variables for this job only
    env:
      GENERATE_SOURCEMAP: false
    steps:
      - uses: actions/checkout@v2
      - name: Build React App
        run: |
          cd example-app
          npm install
          npm run build
      - name: Generate build artifact
        run: tar -czvf example-app.tar.gz example-app/build
      # Upload the artifact so it can be used by future jobs
```

```
      - name: Upload build artifact
        uses: actions/upload-artifact@v2
        with:
          name: react-build
          path: example-app.tar.gz

  deploy:
    # Only deploy if the code was commited to master, not for PRs
    if: github.ref == 'refs/heads/master'
    needs: build-react
    # Note no container is specified, this job will run in the shell
    runs-on: [self-hosted, deploy]
    env:
      PORT: 8081
    steps:
      - uses: actions/checkout@v2
      # Grab the build artifact from the previous job
      - name: Download build artifact
        uses: actions/download-artifact@v2
        with:
          name: react-build
      # Recreate the Docker container(s) running our deployment
      - name: Redeploy
        run: |
          tar -xvf example-app.tar.gz
          docker-compose -p github up --force-recreate -d
```

3.3 GitLab Configuration

GitLab also uses a YAML configuration file to manage CI/CD pipelines. While GitHub supports having multiple files so individual workflows can be separated, all of GitLab's configuration is stored within a single `.gitlab-ci.yml` file; the documentation for it can be found here. The following configuration does the exact same thing as the previous GitHub configuration and is also heavily commented:

```
# Define the three stages that will be run, in order
stages:
  - test
  - build
  - deploy

# GitLab allows variables to be defined and used across multiple jobs
variables:
  DEPLOY_BRANCH: "master"
```

```yaml
test react:
  # Part of the test stage. Multiple jobs can be part of one stage.
  stage: test
  # Run within Docker using the `node:14` image
  image: node:14
  # This job will run on any runner with the tag "docker"
  tags:
    - docker
  # Only run if the branch has an associated merge request
  # or if the commit was made to the deployment branch
  rules:
    - if: '$CI_MERGE_REQUEST_ID != null || $CI_COMMIT_BRANCH == $DEPLOY_BRANCH'
      when: always
  # these commands are run as part of the set up for the job
  before_script:
    - cd example-app
    - npm install
  # the commands that are a part of the job itself
  script:
    - npx prettier --check .

build react:
  stage: build
  image: node:14
  tags:
    - docker
  rules:
    - if: '$CI_MERGE_REQUEST_ID != null || $CI_COMMIT_BRANCH == $DEPLOY_BRANCH'
      when: always
  # Specify environment variables for this job only
  variables:
    GENERATE_SOURCEMAP: "false"
  before_script:
    - cd example-app
    - npm install
  script:
    - npm run build
  # commands to run once the job has finished
  after_script:
    - tar -czvf example-app.tar.gz example-app/build
  # these artifacts can be accessed in later stages of the pipeline
  artifacts:
    paths:
      - example-app.tar.gz
```

```
deploy:
  stage: deploy
  # Run in the shell, not in Docker
  tags:
    - shell
  rules:
    # Only deploy if this is a commit to the deployment branch
    - if: '$CI_COMMIT_BRANCH == $DEPLOY_BRANCH'
      # Only run job if previous jobs were successful
      when: on_success
  variables:
    PORT: 8082
  # Note that the "build" stage artifact is automatically available
  script:
    - tar -xvf example-app.tar.gz
    - docker-compose -p gitlab up --force-recreate -d
```

3.4 Commit and Push

Once all of the configuration files are ready, simply commit and push.

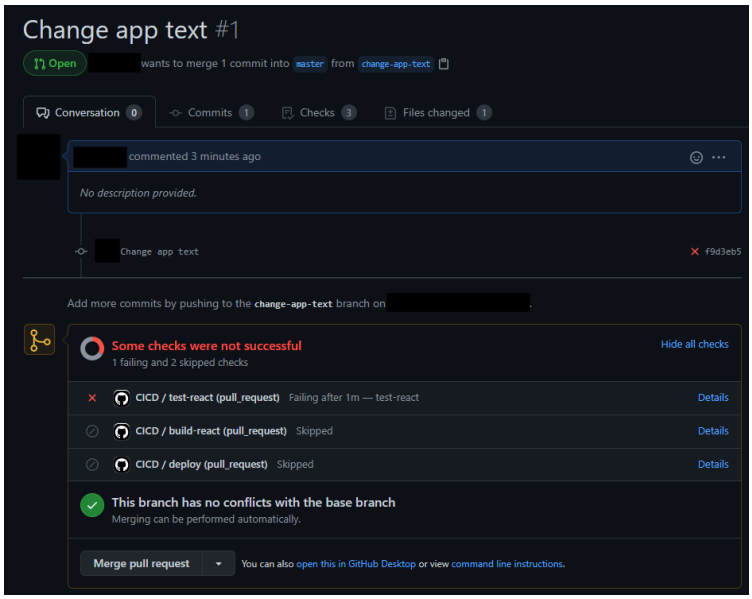
4 Running the Pipelines

This is the easy part - your pipelines have already run! If GitHub or GitLab detect a configuration file, they will read and run it automatically. In a few minutes you should expect an email saying that your pipelines have failed (assuming your notifications are on). They failed because our `test` step runs Prettier to enforce code formatting but we never actually formatted our code.

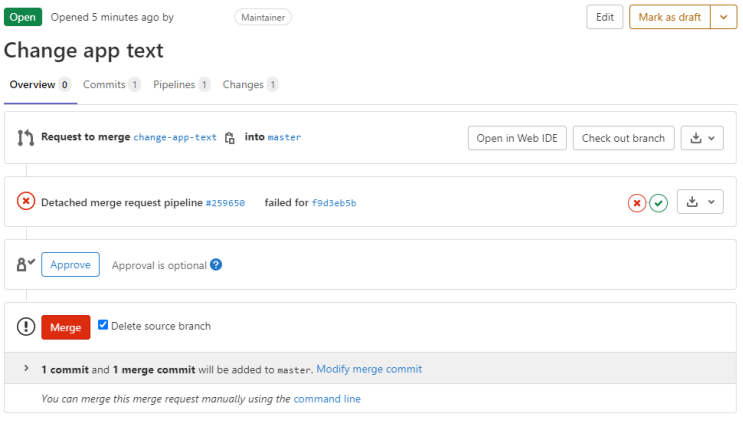
To simulate how the pipelines work with a proper Git workflow, we'll create a separate branch to change a bit of the app's text and fix the code formatting in the process:

1. Checkout a new branch: `git checkout -b change-app-text`
2. Edit some text in `example-app/src/App.js`
3. Commit and push
4. Create a PR on GitHub and an MR on GitLab

After creating the PR & MR, both GitHub and GitLab should automatically run the applicable jobs. Once they complete running, the first job will fail:



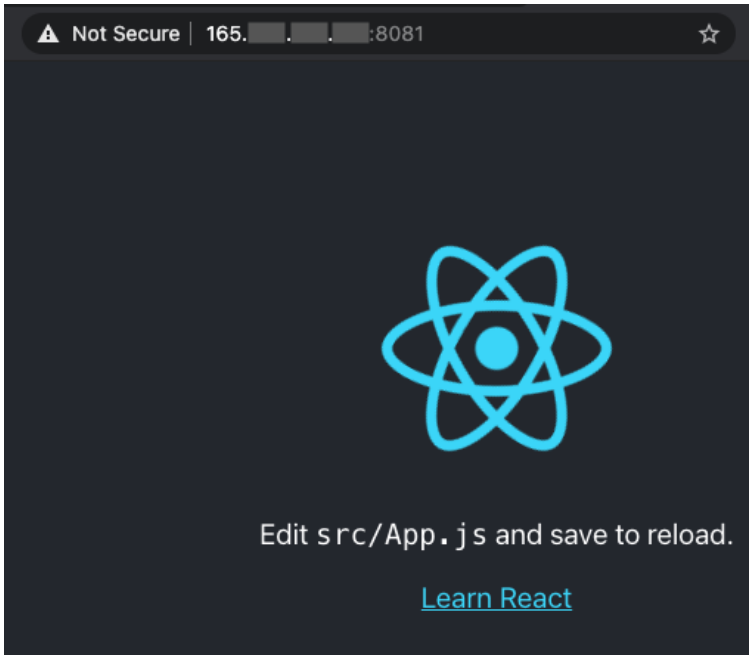
GitHub



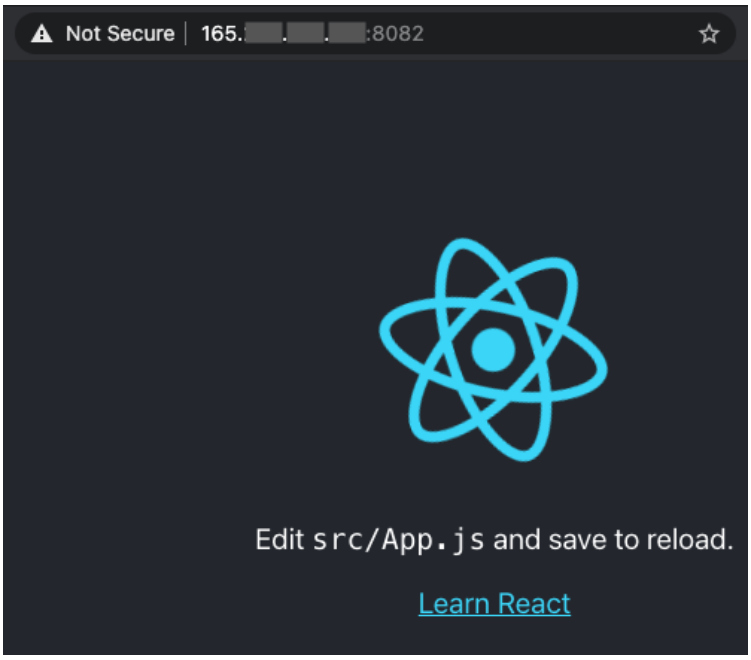
GitLab

GitHub and GitLab can both be configured to disallow merging if some of the jobs fail (not shown above). In order to fix these failures, in the `example-app` folder run `npx prettier -w .` and then commit and push again. The PR & MR will automatically re-run the jobs and they should succeed this time. Once merged, the jobs will automatically run again and this time actually deploy our app!

The ports used to access the deployments correspond to the `PORT` environment variable set in the configuration files above:



GitHub deployed to port 8081



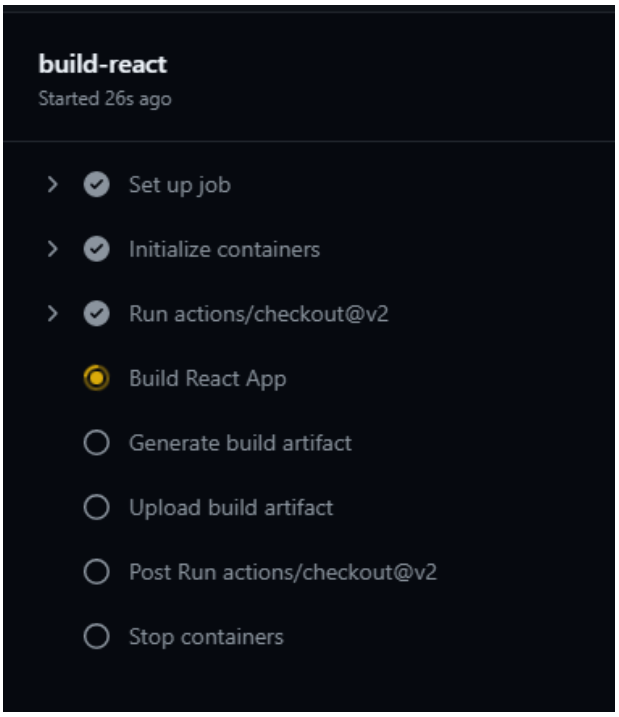
GitLab deployed to port 8082

5 Review of Each

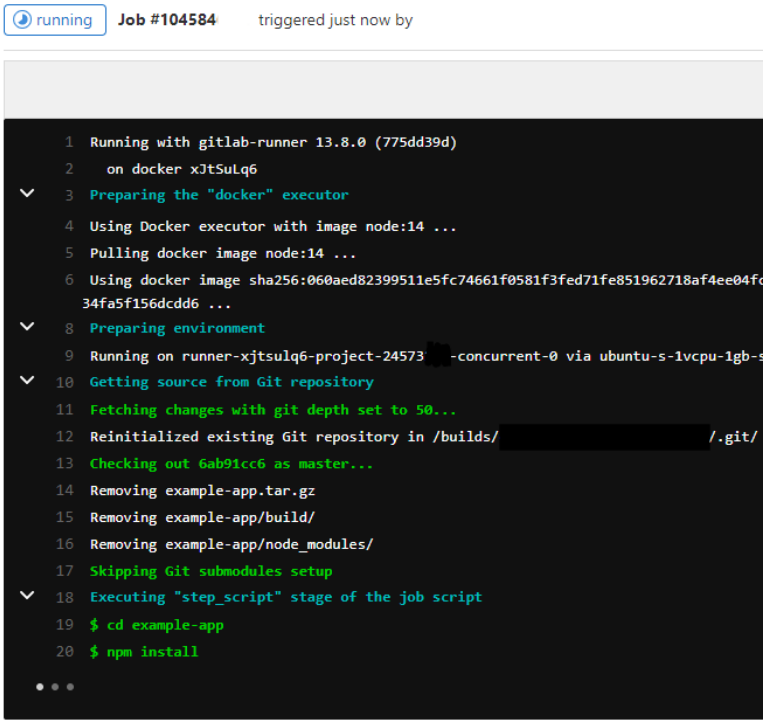
5.1 GitHub

GitHub Actions is extremely flexible and provides lots of options, many of which were not covered here. GitHub also provides more free hosted minutes than GitLab, which is much more convenient than setting up a self-hosted runner (and, also, free). Their self-hosted runner works well, but is quite cumbersome to set up when compared with GitLab's runner. In particular, it is non-obvious how to set up multiple runners and the runners are not installed as a service by default - they must be either manually turned on or manually installed as a service. Generally I believe most developers would expect them to be running all the time (otherwise they cannot receive and run jobs). That being said, once it is set up it does seem to "just work".

An additional issue I noticed was I was not consistently able to see job output while the job was running. It worked some of the time (albeit rarely), although I was consistently able to see it after the job had finished. I have not experienced this issue on GitLab. This is what each platform shows while the jobs are running:



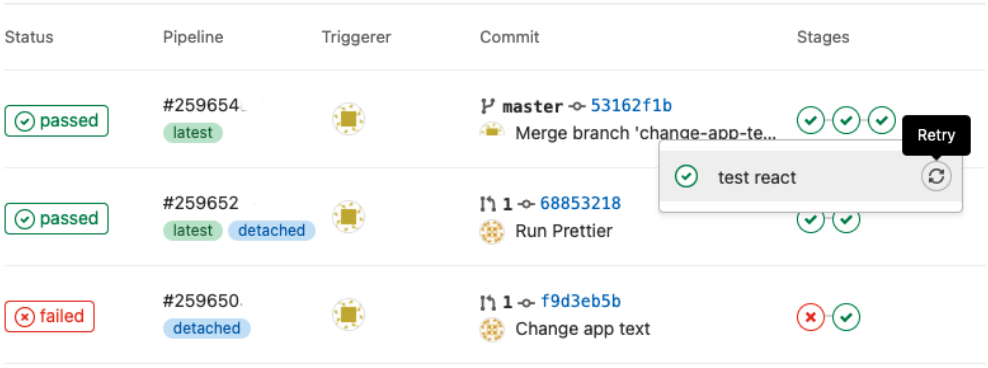
Output of "Build React App" not shown (GitHub)



Full output shown (GitLab)

5.2 GitLab

GitLab has a solid CI/CD offering that is also incredibly configurable. Their self-hosted runner is very easy to set up and works quite well. Additionally, GitLab provides a much nicer interface to see all previous pipeline runs and allows rerunning specific jobs (whereas GitHub requires the entire workflow to be run again):



One drawback of GitLab's approach is all jobs are contained in one file, therefore with many jobs it may become difficult to maintain (whereas GitHub allows multiple "workflow" files). I also noticed that when using `gitlab.com`, CI/CD jobs take a while to start, sometimes up to a minute (despite the fact that they are running on a dedicated self-hosted runner). I have not experienced this issue with self-hosted instances of GitLab nor with GitHub.

5.3 Closing Thoughts

I personally prefer GitLab CI/CD over GitHub Actions for a few minor "quality of life" reasons: their self-hosted runners are easier to set up and maintain, output can be seen while a job is running and it is possible to rerun specific jobs. Since GitHub Actions is newer, I expect that some of these features will likely come in the future. Additionally, GitLab is smart enough to hide jobs that do not match the job's `if` statement, whereas GitHub shows them as "Skipped". I prefer the former (it looks weird to see a skipped `deploy` job for a pull request), although some may prefer the latter.

Nevertheless, both GitHub and GitLab have great CI/CD functionality built in and neither is objectively "better" - try both and choose the one you prefer.