accusoft®

How PrizmDoc Editor Uses GitLab CI

Kyla Kolb
Software Development Engineer in Test (SDET) III

Here at Accusoft, each product has its own, special solution for continuous integration and, in some cases, continuous deployment. One of our newer products, PrizmDoc Editor, has taken a new approach to this and utilized the GitLab CI tool built into GitLab. This new approach has allowed us to have an amazingly small turnaround time for fixing product bugs and issues and getting a better version to the customers in far less time. For this product, GitLab CI was an obvious choice due to our monorepo and already having our repositories hosted in GitLab. This being our first product to use the GitLab CI tool, the structure put into place for the PrizmDoc Editor project has also been a guideline for some of our other PrizmDoc products as well.

GitLab CI Background Information

For those unfamiliar with GitLab CI, it is a tool that is fully integrated into GitLab for the purpose of continuous integration, continuous delivery and continuous deployment. The premise is that every time code is checked in, a pipeline of scripts is automatically run to build, test, and validate the code changes before merging them into master. For more information, check out the marketing website and documentation.

Essentially, what you need to know is that in order to run tests through GitLab CI, you need to have at least one GitLab instance and at least one GitLab runner. A pipeline is split into multiple stages and each stage has one or more jobs. Jobs are the collection of instructions outlined in a .gitlab-ci.yml file that the GitLab runner executes in order to build, test, or deploy code. Stages are gated when run automatically (the next stage does not start executing until the previous one has finished and passed) unless otherwise specified. All jobs that belong to a single stage execute in parallel when that stage is running.

PrizmDoc Editor GitLab CI Pipeline Structure

Our GitLab runner is shared between multiple projects and exists on an always-on Ubuntu 18.04 virtual machine that uses docker-machine to spawn up to ten Ubuntu 18.04 Openstack instances with docker installed on them.

There are two different pipelines for the product: the manual pipeline and the master pipeline. The manual pipeline is the one available when running pipelines against individual commits to branches, which are not master, or merge requests. The master pipeline is the pipeline that will automatically start upon the acceptance of every merge request that is merged to master.

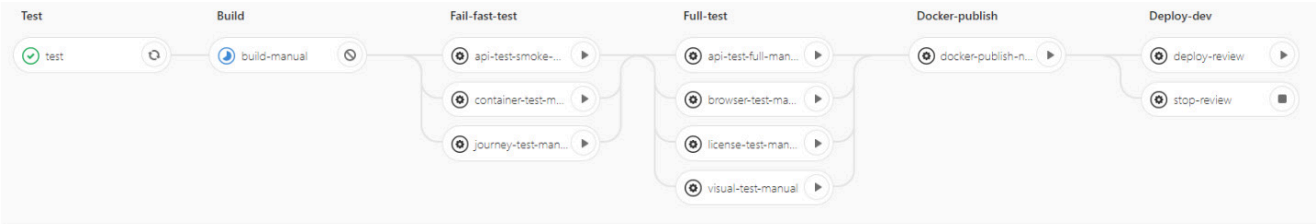
Manual Pipeline

The manual pipeline is associated with all commits that are not associated with the master branch. The pipelines are separated from each other by using the `except` and `only` keywords in the .gitlab-ci.yml file. The jobs that belong to the manual pipeline get the keyword `except: - master` and the jobs that belong to the master pipeline get the keywords `only: - master`. This tells the runner to not add the manual jobs to the master pipeline and that the master jobs only belong to the master pipeline:

```
test-manual:
 <<: *test-job
 when: manual
 dependencies:
   - build-manual
 except:
   - master

test-master:
 <<: *test-job
 dependencies:
   - build-master
 only:
   - master
```

There are six stages in the manual pipeline: **Test, Build, Fail-fast-test, Full-test, Docker-publish,** and **Deploy-dev:**



When code is pushed to a branch or merge request in the remote repository, the Test stage automatically runs. The **Test** stage is the only stage in the manual pipeline that runs automatically and is also included in the master pipeline, therefore it does not get the keyword `except: - master`. Subsequently, all of the other manual pipeline jobs have `when: manual` specified in the gitlab-ci.yml file to let the runner know that these stages do not run automatically. Because of this, a developer can run any test job in the manual pipeline in any order they wish, barring dependencies on other jobs:

```
test-manual:
 <<: *test-job
 when: manual
 dependencies:
   - build-manual
 except:
   - master
```

Stage: Test

In the **Test** stage, there are test jobs which include unit tests and other quick tests. We have approached our testing from a fail-fast mentality by running these two test suites before building the product because neither one of these requires the product to be built, and they both run in under 10 minutes. The **unit** job runs the Jest unit tests and junit code coverage report in a node-based docker container. The job will fail if code coverage drops under the set thresholds in our .jestrc.json file.

Stage: Build

While this stage must be run manually and any other jobs or stages can be manually run before/without the **Build** stage finishing, those jobs will fail, as the rest of the manual pipeline depends upon the **Build** stage. The stage is not set to be automatic in order to save on time and resources since the pipeline runs on every commit push to a branch or merge request and not every commit needs more testing than the **Test** stage. It is noted in the gitlab-ci.yml file that the other jobs have a dependency on the **build-manual** job within the **Build** stage:

```
test-manual:
 <<: *test-job
 when: manual
 dependencies:
   - build-manual
 except:
   - master
```

The **Build** stage contains only one job: **build-manual** and takes approximately 15 minutes to run. This job runs the commands necessary to build the product in a Docker-in-Docker (dind) based container and keeps as artifacts the docker image, MySQL scripts, and version to be used in later jobs. Because the only differences between the **build-manual** job, that belongs to the manual pipeline, and the **build-master** job, that belongs to the master pipeline, are keywords, the two use a common hidden job that is denoted as such by a prepending period. The prepending period ensures that the scripts will not be processed by GitLab CI as a job, but they can still be referenced and run using YAML features such as anchors (&), aliases (*) and map merging (<<).

In the below code example, you can see the hidden job (**.build**) is anchored as **build** and the two separate build jobs (**build-master** and **build-manual**) use map merging and aliases to refer back to the hidden job. You can also see that even though the two use the same base scripts, the **build-master** job is only on the master pipeline and the artifacts expire after 1 week. In comparison, the **build-manual** job is triggered manually, not in the master pipeline and the artifacts expire after only 12 hours. The difference in the artifact time is due to the volume of manual pipelines versus master pipelines as well as the usefulness of the artifacts from a changing branch or merge request pipeline, versus a master pipeline:

```
.build: &build
 stage: build
 tags:
  - tag
 Image: image
 cache:
   key: cache_key
   paths:
     - path/to/something/
 script:
  - list of commands to execute in docker container
  - more commands

build-master:
 <<: *build
 only:
   - master
 artifacts:
   paths:
        path/to/something/
   expire_in: 1 week

build-manual:
 <<: *build
 when: manual
 except:
   - master
 artifacts:
   paths:
   - path/to/something/
   expire_in: 12 hours
```

Stage: Fail-fast-test

The **Fail-fast-test** stage is another look into our approach at failing fast. The jobs in this stage run the quickest, are the most stable, and have the largest variety of tests that we have. This stage is similar to a smoke stage and will allow us to know very quickly that there is something wrong with the commit being tested. The stage takes around 12 minutes to run and consists of jobs ranging from a smoke api test suite to a container-based test suite, to a product-level browser journey test suite.

All of these jobs are set up similarly to the **Build** stage as dind containers that use a series of map merged hidden jobs leading back to two common hidden jobs: **docker-test-setup** and **test-script (test-script** being used in the `before_script` for each job). The api and journey tests also share common hidden jobs that specify the configuration of the product.

The main differences between the jobs at this stage are how the tests are run, the image for the container they use in the scripts, and when the artifacts are saved. For the api and journey tests, the artifacts are saved when encountering failure. For the docker container tests, the artifacts are always saved. For all jobs, the artifacts expire after one week.

```
.container-test: &container-test
 <<: *docker-test-setup
 stage: fail-fast-test
 before_script:
   - *test-script
 script:
   - test script
 artifacts:
   when: always
   paths:
   - path/to/something/
   expire_in: 1 week

.docker-test-journey: &docker-test-journey
 <<: *docker-test-setup-configuration
 stage: fail-fast-test
  script:
   - test script
 artifacts:
   when: on_failure
   paths:
     - path/to/something/
   reports:
     junit: 'test/app/test/journey/test-results.xml'
   expire_in: 1 week
```

Stage: Full-test

The **Full-test** stage is the longest-running stage we have, averaging around 45 minutes of run time. The jobs include a full api test suite, more extensive product-level browser tests, licensing tests, and visual regression tests.

The jobs in this stage are set up similarly to those in the **Fail-fast-test stage** and have common hidden jobs that they all use, all keep their artifacts `on_failure`, have differing images for the test containers, and have differing ways of calling the actual test script.

Stage: Docker-publish

The **Docker-publish** stage is used in the manual pipeline to publish the built container. In the manual pipeline, this stage has one job that publishes the built docker container to our internal Nexus server for use in our development environments. This job uses a dind container that builds and pushes another container based off of the artifact from the **Build** stage. Because there is also a **docker-publish-master**, this job also uses a common hidden job. This stage takes about a minute to run.

Stage: Deploy-dev

Deploy-dev is a stage used for deploying a review app for a specific commit or merge request. Review apps are useful when testing out the product and needing an environment that is not your local machine, as well as providing a common place for other developers to manually test and review the application. There are two jobs in this stage: **deploy-review** and **stop-review**. This stage works in accordance to the GitLab built-in review apps and dynamic environment capabilities. It follows the guides of the review app documentation in addition to running the scripts to start the product docker container. **deploy-review** and **stop-review** also both map merge to a hidden job, **review-stop-script**, that is used to stop the product and clean up the container.

```yaml
deploy-review:
 stage: deploy-dev
 tags:
   - review
 when: manual
 except:
   - master
 dependencies:
   - build-manual

 environment:
   name: review/$CI_COMMIT_REF_NAME
   url: url
   on_stop: stop-review
 before_script:
   - *review-stop-script
 script:
   - start product

stop-review:
 stage: deploy-dev
 dependencies: []
 tags:
   - review
 when: manual
 environment:
   name: review/$CI_COMMIT_REF_NAME
   action: stop
 script:
   - *review-stop-script
```

Pipeline Failures

When one of the jobs fails in the manual pipeline, the stage and job will show an icon of an orange (!):

Failing jobs do not gate other jobs in the manual pipeline and no notifications are sent. All jobs can be re-run any number of times by clicking the refresh button on the right side of the job title. This is helpful if the failure is a flaky product, bug, or test infrastructure issue and you can experiment with how often the job fails.

Master Pipeline

The master pipeline is associated with all accepted merge requests into master. As there should be no direct pushes to master, this is the only time that the master pipeline will fire. Our process is to run the full manual pipeline on the merge request that we are reviewing before pressing the 'merge' button. Once the button is pressed, the code is merged to master and the master pipeline begins. The full pipeline takes approximately two hours, depending on resource availability and pipeline flakiness.

The structure of the master pipeline is largely the same as the manual pipeline with just a few differences. The main difference is that the entire pipeline is automated and gated on the completion of each stage. For example, once the Test stage runs and passes, the Build stage will automatically start and when that passes, the Fail-fast-test stage will start and so on. The order of the stages is set at the beginning of the .gitlab-ci.yml file:

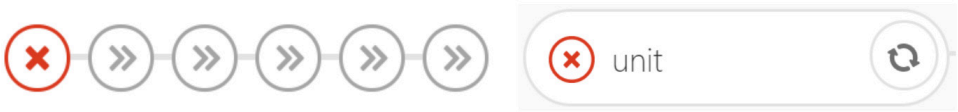
```
stages:
 - test
 - build
 - fail-fast-test
 - full-test
 - docker-publish
 - deploy-dev
 - notify-fail
```

The first four stages in the master pipeline are the same as the manual pipeline with the exception of them being automatic. These stages are **Test, Build, Fail-fast-test**, and **Full-test**. The **Docker-publish** stage now has added jobs that publish the docker container publically.

The **deploy-dev** stage is different as well. Instead of a review app, this is a regular job that only runs on master using `only: - master` and deploys the product to our internal development environment. This is different from the review app as it is an always-up instance of the latest version of our product for development and testing purposes.

Finally, there is now an additional stage **Notify-fail**. This stage contains one job, **notify-failure**, that only triggers when one of the other stages in the pipeline has failed. As soon as a job has finished and reported back that it has failed, the pipeline will finish up all the jobs in the stage that it's on and then skip all of the other stages to run the **Notify-fail** stage. A notification is then sent to our internal Slack channel that the master pipeline has failed. At this time, we can go investigate the failure and either roll back the changes or re-run the failed stage/job to assess pipeline flakiness.

The failed stage and job are denoted by a red (x):



The stage-skipping is accomplished by adding `when: on_failure` to the top sub-level of the notify-failure job in the .gitlab-ci.yml file:

```
notify-failure:
   stage: notify-fail
   image: $NODE_IMAGE
   tags:
     - docker
   only:
     - master
   when: on_failure
   script:
     - notify failure script
```

Closing Words

So, there you have it. How PrizmDoc Editor is currently using GitLab CI to continuously integrate and deploy our product. This pipeline has streamlined our development cycle and aided us in becoming more agile with the way that we test and deploy the product. However, this pipeline is never considered complete and is ever-evolving. We are constantly perfecting it and adding new suites as we go along. We are currently working on a solution for getting Windows-based testing into the GitLab CI pipeline which is, perhaps, a subject for another article. I hope that this article and the result of our experiences with GitLab CI have given you an idea of the power that a good pipeline can have.