2DV611 - Continuous Delivery

Alex Karlsson
Olof Haga
Jesper Eriksson
Marcus Cvjeticanin

1. The application

Introduction

Loggen is a web application for collecting all your notes sorted after a time frame. You can save logs like when you had your last meal or when you went to bed last night. Loggen provides a clearer time table of when you perform certain tasks.

The application is written in *JavaScrip*t using the frameworks *Node.js*, *Express* and *Reactjs* and with *MariaDB* as an underlying database. The application is deployed in a *Kubernetes* cluster that is hosted on *OpenStack*. Storage is handled by a *NFS* server also hosted on *OpenStack*.

The application is developed after a continuous delivery philosophy which includes utilization of a deployment pipeline integrated with GitLab which is used as version control in this project.

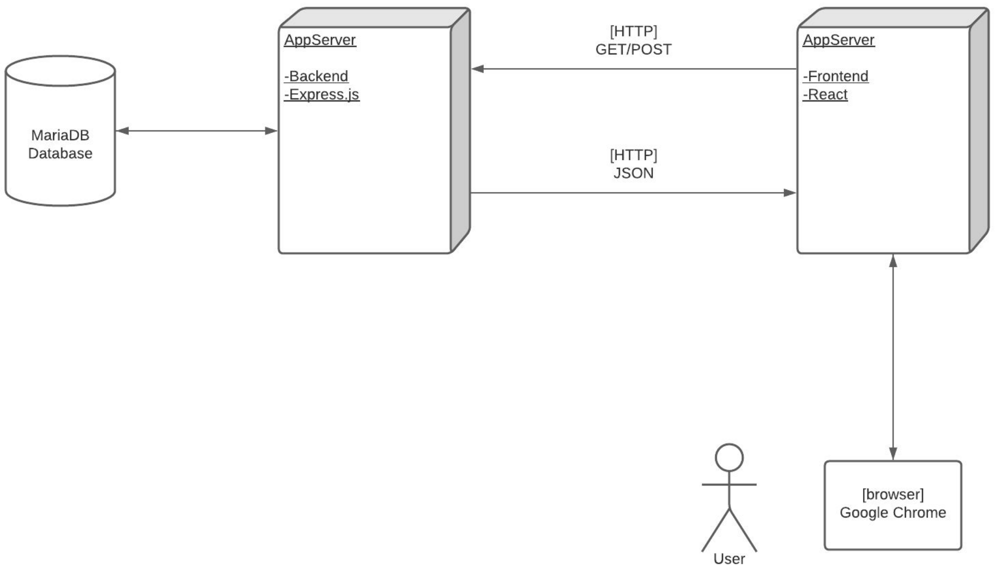
Repository

The project repository with all its artifacts, including *Ansible Playbook* script for infrastructure provisioning, codebase, tests, configuration files and image repository can be below:

https://gitlab.lnu.se/oh222fv/2dv611-team2

Architecture

The application has a *3-Tier Architecture* that consists of a frontend, backend and database tier. This separation into several tiers provides us with several benefits such as scalability. Since the deployment makes use of containerization and *Kubernetes* with the option of scaling up the amount of replicas this can also help make the application more resilient as there should be other containers available on failure. As more features are added to the application it is possible to expand this architecture into an *N-Tier Architecture* that is divided into more tiers.



Frontend

For the frontend we use the *React framework* and use *React Hooks* as opposed to classes. The library *React Hook Form* is used for form validation and error handling. The application has protected routes and *JWT tokens* are used for authentication and authorization [1]. The in-built *React Context* is used to manage authentication and users state. When requests to the backend are made the *JWT tokens* are sent as headers for further validation and authorization.

Backend

For the backend we use *Node.js* with *Express*. The backend is a REST API that communicates to a *MariaDB* database. We are using route/controller/service/database structure in the backend. The database folder contains *DAO* (*Database Access Objects*) for interacting with the database. This structure makes it simpler to control the requests and to provide the correct service for each request. This also makes it easier to replace components with another implementation, like for instance replacing *MariaDB* with another database. For more information concerning the architecture of the backend please refer to the codebase itself found in the Repository section of this document.

Development environment

Important for a new developer to know is that *Docker* Compose is used in development to build, start and stop the application. Each tier in the 3-Tier Architecture (frontend, backend and database) is defined as a service in the *docker-compose.yml* file. It also includes an additional service called *Adminer* which is a database management tool that can be used to manage the data in development. It also provides an interface that can be used to test queries and to perform operations on the data in the database.

Docker is all that should be required in order to build, start and stop the application:

- docker-compose up
- docker-compose down
- docker-compose build

Documentation about other *Docker Compose* commands can be found here: https://docs.docker.com/compose/reference/

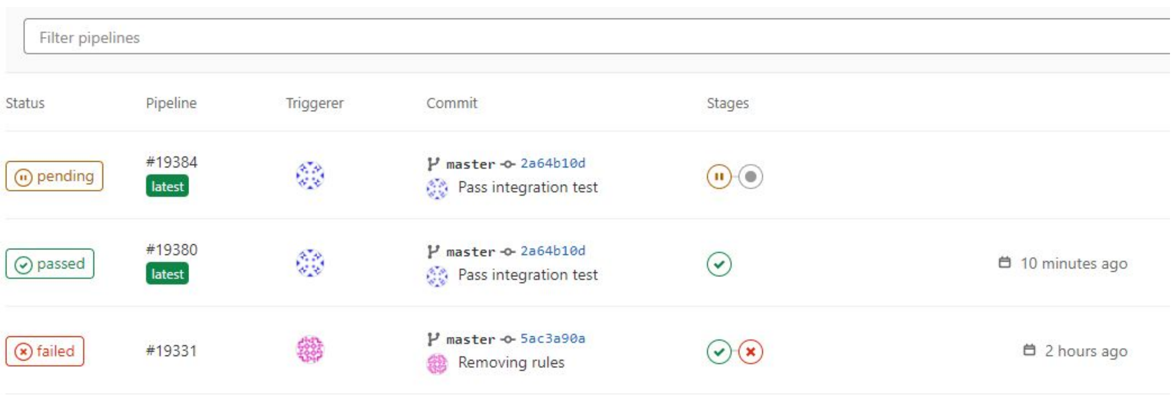
Environment variables are also set with *Docker Compose* (either with a .env file or directly in the *docker-compose.yml* file). The application makes use of environmental variables for repetitive, dynamic or sensitive information that should not be included in version control, e.g., for the database connection in the backend.

Bind mounts are used for live reload and local volumes are created to persist data in development.

Linting is also used in development to help ensure that the code is consistent and readable.

2. The Pipeline

Note that some GitLab features used in our pipeline are described in more detail under section 3.



Filter pipelines

Status	Pipeline	Triggerer	Commit	Stages	
pending	#19384 latest		master 2a64b10d Pass integration test		
passed	#19380 latest		master 2a64b10d Pass integration test	✓	10 minutes ago
failed	#19331		master 5ac3a90a Removing rules	✓ ✗	2 hours ago

Stages

The following stages that are used for our pipeline are:

- **build** - On the build stage we build by using the *Node.js npm install* to install the necessary packages for the application that will be used.

- **lint-test** - We run the lint to discover any errors of the linting.

- **unit-test** - Run all the unit tests.

- **coverage-test** - Run coverage test.

- **build-image** - On this stage we build the *Docker* image and push the image up to the registry.

- **deploy-staging** - Deployment of the image to the *Kubernetes* cluster on the staging environment.

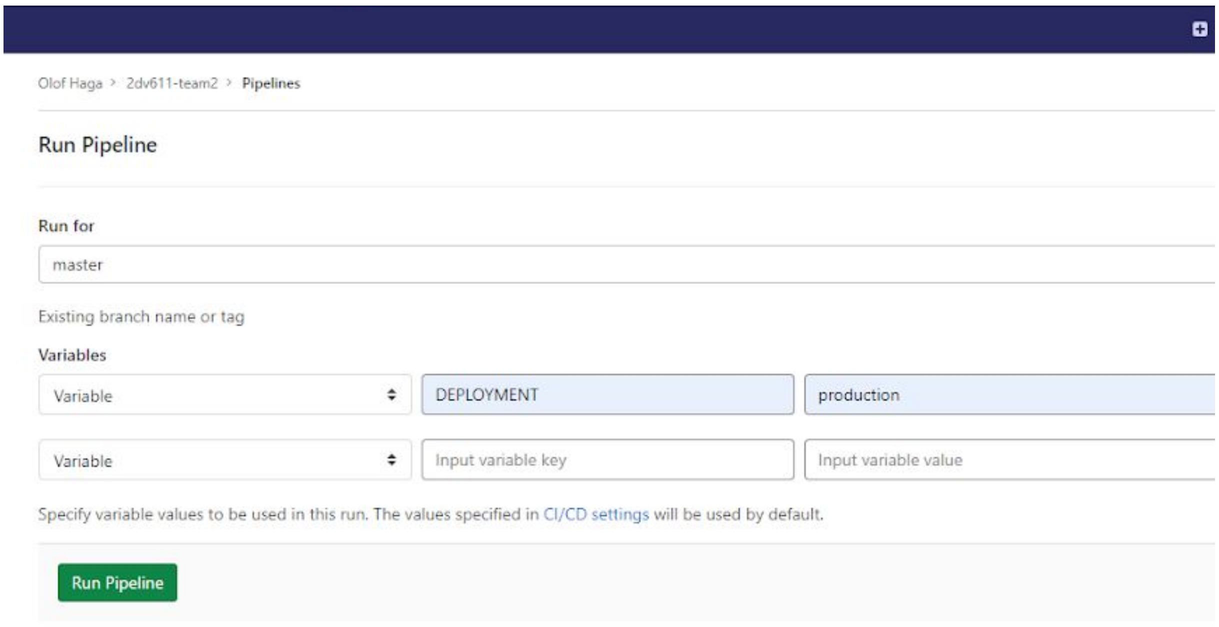
- **integration-test** - On this stage we do some integration tests with *Selenium*.

- **deploy-production** - If the integration test works we deploy the image to the Kubernetes cluster on the production environment.

- **smoke-test** - Finally we do some smoke tests with *Selenium* to see if the application is running in the *Kubernetes cluster*.

Deployment

We use *Continuous Delivery* for releasing the application to production with *Kubernetes*. This is a manual step in which the pipeline runs with the variable DEPLOYMENT set to production:


Olof Haga > 2dv611-team2 > Pipelines

Run Pipeline

Run for

master

Existing branch name or tag

Variables

Variable	DEPLOYMENT	production
Variable	Input variable key	Input variable value

Specify variable values to be used in this run. The values specified in CI/CD settings will be used by default.

During normal development when code is merged into the master (without the variable DEPLOYMENT set to production) the application is deployed into staging.

As a side note, removing this variable from the pipeline related files would result in *Continuous Deployment*. However, this is not something that we intend to use today.

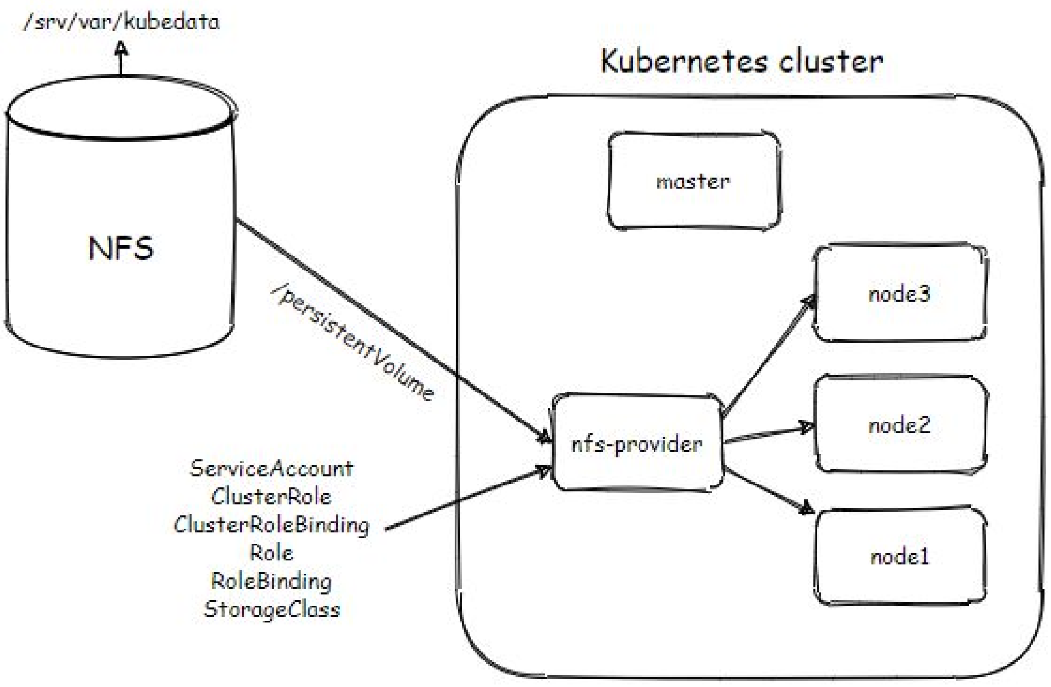
The pipeline is divided into two different child pipelines (backend, frontend). They are configured so that if changes are made to frontend it will only run the frontend child pipeline. Note that if you run the pipeline manual (which is necessary when deploying to production) it will run both the frontend and backend pipeline even if no changes have been made.

Persistent data

Since we make use of a *MariaDB* database we also need to implement persistent volume. Persistent volume allows us to maintain data created in the database in case the database would go down or the node which runs the *MariaDB* deployment needs to be restarted or goes down. However there are different ways of setting up persistent volumes in *Kubernetes*. You could store the data in one of your nodes and then mount the database deployment from there.

There are some downsides to this approach though. First, this requires us to use disk space on our nodes to store data. Secondly, this approach will lose all data in case of the cluster going down. A persistent volume lives inside the *Kubernetes* etcd core which means that if the cluster goes down the volume and data will go down with it. Therefore, we chose to use an external provider of persistent volumes. This means that we have an external source of storage such as a file server. Many traditionally used cluster providers such as *GKE* (Google) and *Amazon EKS* have a built in service that provides this functionality. We chose to implement our own since *Cinder* (which is *OpenStacks* block storage service) was deprecated in *Kubernetes* official documentation. We used a network file system (*NFS*) server to provide volumes for us. We also implemented this in a dynamical way which means that we do not have to create the volumes ourselves but the *NFS* does that for us as well. So, with our solution we only (once) need to set up a *NFS-provide*r deployment with belonging *ServiceAccount*, *Role*, *RoleBindings*, *RlusterRole* and *ClusterRoleBindings* to allow it to create persistent volumes for us whenever we need it.

This allows us to build *Deployments* with persistent volumes through the pipeline without having a cloud administrator manually create a *Volume* for the *Deployment* to claim, now we only need to claim storage space through *PersistentVolumeClaims* and the *NFS* server will provide the volume for us. The image below shows how the environment is set up with the *NFS-provider*.



Exactly how the nfs-provisioner is set up can be viewed in the master node at ~/nfs-provider/. There we have 3 filles. *RBAC.yaml* which sets up the needed roles and bindings for the *NFS* provisioner to get the correct accesses. *Class.yaml* which sets up the *Storage class* that refers to the nfs server. *Deployment.yam*l which deploys the nfs-provider itself. The *NFS* server is currently set up in a different *OpenStack* project. You can access it by *194.47.177.56*. Information about how the deployment of mariadb is set up can be viewed in the *GitLab* repository in the *Kubernetes* folder.

Testing

Unit tests

For unit testing on the backend we are using the test framework and *npm* module *Mocha* [2] together with the npm module *Chai* [3]. Running the command *npm run test* will execute all tests located in */backend/tests*.
The tests implemented until today focus on testing the validation methods used for registering new users.

For unit testing on the frontend we are using the test framework and npm module *Jest* [4] along with *React* to make test cases. Running the command *npm run test* will execute all tests that are located in folders name __*tests*__ with the file extension **.test.js*.
The tests implemented until today focus on testing the *React* components to see that they are rendered correctly.

Lint tests

The codebase in both backend and frontend are validated with linting using the npm module *eslint* [5]. Running the command *npm run lint* works in both backend and frontend and will lint the code. The linting configuration can be found in the file *.eslintrc.json* which can be found in the root directory of the both components.

Code coverage

Code coverage is until today only implemented on the backend. Here we are using the npm module *nyc* [6] together with *Mocha* to test how much of the codebase that is covered by the unit tests. Running the command *npm run coverage* will start the code coverage testing.

Integration tests

For the integration testing we are using the web application testing framework *Selenium* [7]. These tests are written in *Python* and are testing the integration between the three components; frontend, backend and database. The tests can be found in the *Selenium* directory in the root of the project and the command *python integration-test.py* are used to run the tests.
Until today we are testing that it works to register a user and is possible to login as that user.

Smoke tests

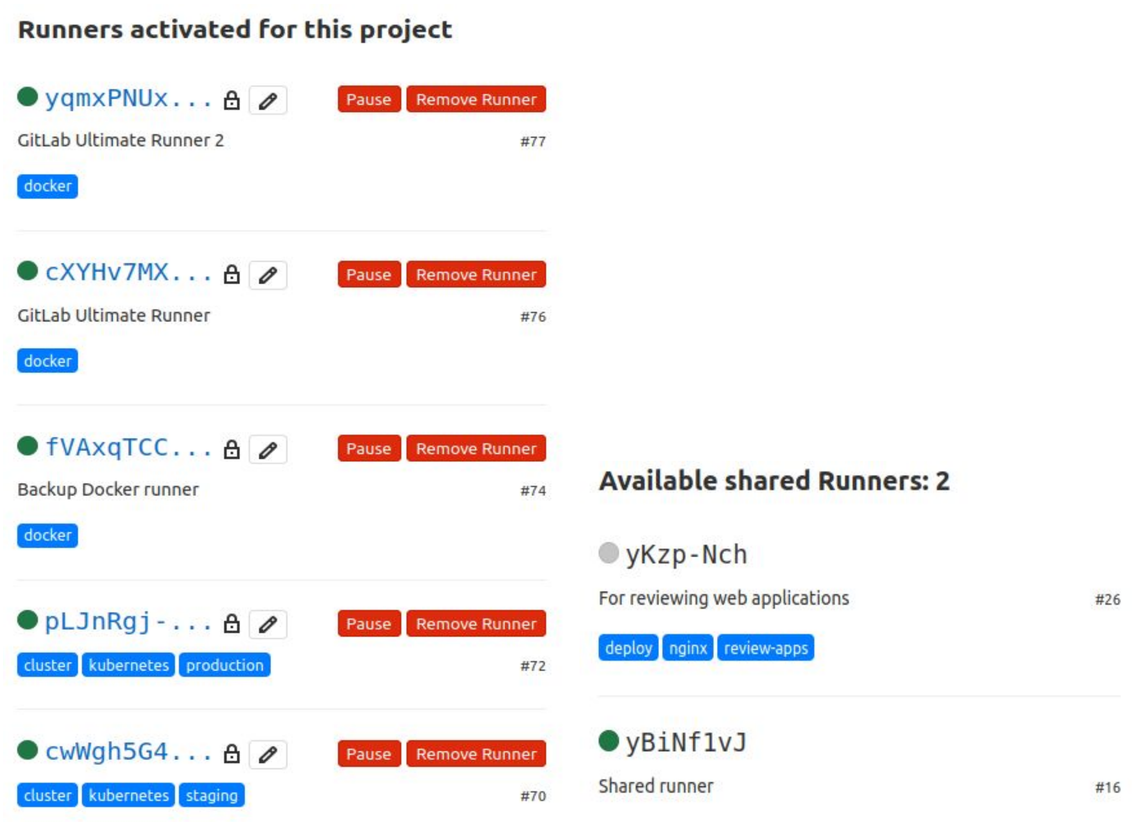
For the smoke testing we choose to also use the web testing framework *Selenium* with *Python* to check if the site is up and running by asserting if the site title is correct.

3. GitLab CI / CD

Runners

The runners that were used were two shared runners and five specific runners with different purposes.
The three runners, that are tagged with *Docker*, are used for the build and test stages and are configured with the *Docker executor*. We created three of these to have some backup if one of them is down but for running the pipeline it's enough to have one runner with the *Docker tag*. Two is preferable though.
The two runners tagged with *Kubernetes* are used for deployment to the clusters in each of the environments used. Staging and Production. One runner is installed in each one of the clusters and is also using the *Docker executor*.
The shared runners are not managed but us, but are used as backup and can be used when our own runners are busy.

Runners:



Runners activated for this project

yqmxPNUx... Pause Remove Runner
GitLab Ultimate Runner 2 #77
docker

cXYHv7MX... Pause Remove Runner
GitLab Ultimate Runner #76
docker

fVAxqTCC... Pause Remove Runner
Backup Docker runner #74
docker

pLJnRgj-... Pause Remove Runner
cluster kubernetes production #72

cwWgh5G4... Pause Remove Runner
cluster kubernetes staging #70

Available shared Runners: 2

yKzp-Nch
For reviewing web applications #26
deploy nginx review-apps

yBiNf1vJ
Shared runner #16

https://gitlab.lnu.se/oh222fv/2dv611-team2/-/settings/ci_

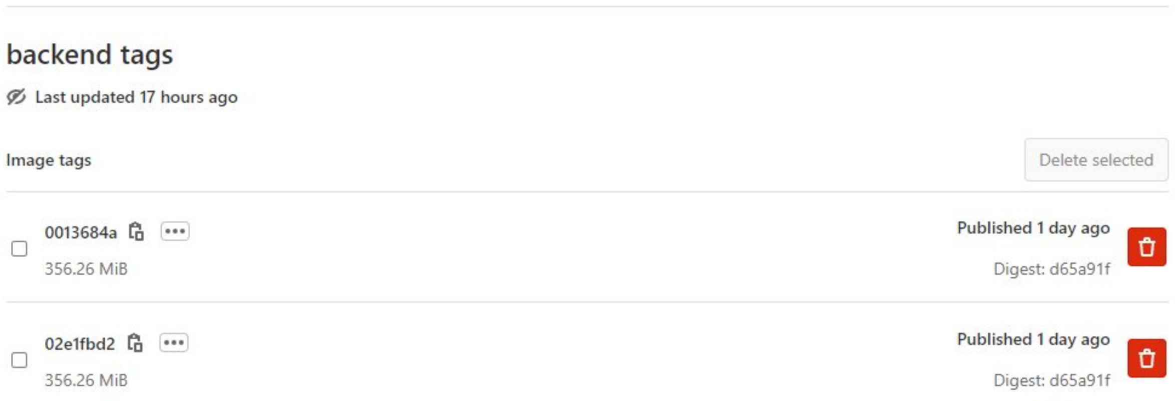
Container Registry

We chose to save our images to the *GitLab Container Registry*. We have three different Image Repositories in our *Container Registry*:

- oh222fv/2dv611-team2/backend
- oh222fv/2dv611-team2/mariadb-staging
- oh222fv/2dv611-team2/frontend

Note that we use an *MariaDB Image* hosted on *Docker Hub* in production: https://hub.docker.com/_/mariadb

The frontend and backend images have dynamic tags that are based on the commit *SHA*. This makes it simple to track each image to a specific commit and pipeline run.



backend tags

Last updated 17 hours ago

Image tags

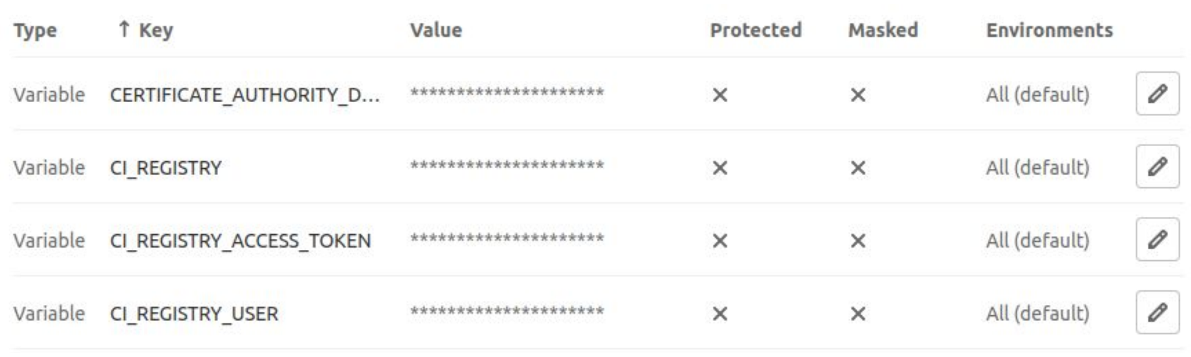
0013684a
356.26 MiB
Published 1 day ago
Digest: d65a91f

02e1fbd2
356.26 MiB
Published 1 day ago
Digest: d65a91f

Access tokens and environmental variables.

We started with project based access tokens that had permissions based on their intended purposes, e.g., pull and push images to the *GitLab Container Registry*. However, we had to switch to personal access tokens instead because of a bug in *GitLab*. More information about this can be found below:
https://gitlab.com/gitlab-org/gitlab-runner/-/issues/27113

The environmental variables that we used was:



Type	↑ Key	Value	Protected	Masked	Environments	
Variable	CERTIFICATE_AUTHORITY_D...	********************	×	×	All (default)	
Variable	CI_REGISTRY	********************	×	×	All (default)	
Variable	CI_REGISTRY_ACCESS_TOKEN	********************	×	×	All (default)	
Variable	CI_REGISTRY_USER	********************	×	×	All (default)	

GitLab Kubernetes integration:

We have connected our staging and production *Kubernetes clusters* to *GitLab* using their *Kubernetes* integration. This allows us to view logs from *Kubernetes* in *GitLab* and it can also help us integrate our cluster with *Prometheus* for monitoring. However, there was a problem with the *Prometheus* integration that we were unable to resolve. This is something that we should look at in the future as *Prometheus* can be used with *GitLab* for automatic rollbacks.

More information about *GitLab's Kubernetes* integration and automatic rollbacks can be found below:
https://docs.gitlab.com/ee/user/project/clusters/
https://docs.gitlab.com/ee/ci/environments/#auto-rollback

Resources to learn more

[1] JWT, “JSON Web Tokens - jwt.io”, https://jwt.io/

[2] Mocha, “Mocha | NPM”, https://www.npmjs.com/package/mocha

[3] Chai, “Chai | NPM”, https://www.npmjs.com/package/chai

[4] Jest, “Jest | NPM”, https://www.npmjs.com/package/jest

[5] Eslint, “ESLint - Pluggable JavaScript linter”, https://eslint.org/

[6] nyc, “NYC | NPM”, https://www.npmjs.com/package/nyc

[7] Selenium, “Selenium Test Framework”, https://www.selenium.dev/