Setting up Gitlab

It is recommended to have at least 4GB Ram available on the machine on which you want to host Gitlab. We decided to host Gitlab on a t2.medium instance on Amazon Web Services (running Ubuntu). It provides the necessary specifications without completely emptying one's wallet. In the process of launching a new instance, you can choose an appropriate AMI of Gitlab. This will basically relieve you of any further installation duties. Under the tag Community AMIs you will find different versions of the Enterprise or Community Edition. We went for the newest version of the Community Edition. It is essential to specify open ports for SSH (22) and HTTP (80). Therefore you need to configure a security group in your AWS interface. Under "Security Groups" you can edit existing groups or add new ones. After making sure, the appropriate ports are opened for inbound traffic, you need to assign the security group to the EC2 instance.

After everything is set up you can access your Gitlab instance via Browser.

Setting up a Runner

Before we are able to use the Runner, which uses the Docker executor, we first need to install docker on our server. SSH into your AWS server and execute the following commands to install docker (updating the apt package index and then install docker-ce).

```
sudo apt-get update
sudo apt-get install docker-ce
```

Now that we have got Docker installed, we can get to installing and configuring the runner.

First we need to add Gitlab's official repository and then install gitlab-runner.

```
curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh \
    | sudo bash

sudo apt-get install gitlab-runner
```

Now we can use the installed CLI to register a new runner, on which the jobs of our CI pipeline are executed. Just execute `sudo gitlab-runner register` and answer the questions of the the command dialog. When asked for the coordinator URL, simply put in the URL of your Gitlab instance. You will find the token for registering the runner under Runner settings inside the CI/CD settings. There is no need to add any tags or set up the runner as unlocked. But all if this can also be changed later in the Gitlab UI. As executor choose docker and as default image choose node. The default image will be used, if there is none defined in the .gitlab-ci.yml file (which we will do anyway).

Defintion of the pipeline

The essential file to specify the continuous integration pipeline is the .gitlab-ci.yml file, which needs to be put inside the root folder of the project. The file serves as the definition of the stages and jobs of the pipeline.

First of all we need to define the docker image, which is used to run the jobs. In our case we want to use the Node.js image, therefore we simply add `image: node` to the yaml file. The ci executor will pull images from the the Docker Hub, so any of the pre-built images can be used here. The described code would use the latest node image, but you may also specify a different version by for example adding

`image: node:8:10:0` instead.

You can also define services (for example databases), which should be used, in the same manner. Any image available on the Docker Hub is possible. Since we use Mongo DB is a database, we add the following to the file.

```
services:
    - mongo
```

Our pipeline consists of 4 stages, which are defined as follows:

```
stages:
    - build
    - test
    - staging
    - deploy
```

The jobs are specified in a similar manner. Th naming of the jobs is arbitrary and up to the developers. If for example we wanted to define a job "build", we would add the following to the yaml file.

```
build:
    stage: build
    script: npm install
```

The stage, in which this specific job is supposed to run, is indicated, as well as the script, which we want to run inside this job. In this case it is fairly simple, as we do not really need to run any complicated build jobs for our Node.js application except for the installation of dependecies. There is a variety of configuration options possible. For an overview of all parameters see https://docs.gitlab.com/ce/ci/yaml/.

Furthermore we want to define two jobs for our test stage, which look pretty similar to one another and deal with linting and testing.

```
run_linting:
    stage: test
    script: npm run lint
    artifacts:
      paths:
        - build/reports/linting-results/

run_tests:
    stage: test
    script: npm run test
    artifacts:
      paths:
        - build/reports/test-results/
```

The code specified for the script parameter, runs the respective npm script in both cases. Those scripts are specified in the package.json file: "lint" triggers the execution of ESLint, whereas "test" uses the Mocha testing framework to run our implemented tests and Mochawesome to create html test reports.

The html pages containing the reports for the linting and testing results are stored in the folders *build/reports/linting-results* and *build/reports/test-results*. That's why we specify those paths in the artifacts parameter, so that gitlab can offer the option to directly download those artifacts.

Finally, we have one job in each of our two stages staging and deploy, which pretty much look the same except for the definition of a different IP address of the appropriate server. The following code defines, what happens in our deployment job.

```
deploy_production:
  stage: deploy
  when: manual
  before_script:
    # Install ssh-agent if not already installed, it is required by Docker
    - 'which ssh-agent || ( apt-get update -y && apt-get install openssh-client -y )'

    # Run ssh-agent (inside the build environment)
    - eval $(ssh-agent -s)

    # Add the SSH key stored in SSH_PRIVATE_KEY variable to the agent store
    - ssh-add <(echo "$SSH_PRIVATE_KEY")

    - mkdir -p ~/.ssh
    - '[[ -f /.dockerenv ]] && echo -e "Host *\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config'

  script:
    - echo "Deploy to production server"
    - ssh deploy@159.89.96.208 "cd /home/deploy/shaky-app && git pull && npm install && sudo
systemctl restart shaky.service"
```

A key difference in comparison to the staging job is the parameter `when: manual`. Although we want to automatically run the job to update the staging server to have the newest code, we do not want the same behaviour for our deployment job. We don't want the job to automatically run, as soon as the ones before are finished, but we only want the job to run upon a manual triggering. Inside of the Gitlab CI interface a "play button" for this specific job shows up, with the help of which the developer is able to manually trigger the deployment. Due to this setup one can check, if the application appropriately works on the staging server, before pushing to deployment.

The code, which is defined in the before_script parameter, is needed, because Gitlab does not support managing SSH keys inside the runner. To deploy we, of course, need to ssh into our server. By adding those commands, we can inject an SSH key into the build environment. For further details see https://gitlab.ida.liu.se/help/ci/ssh_keys/README.md.

The actual command (inside script) to deploy our code simply uses SSH to access the deployment server, where we have already set up a connection to the git repository. Now we only need to pull the current code base, update the dependecies (npm install) and restart the service. This works the same for the deployment to the staging server.