Deploy Innogle application using Docker compose.
Project Report

Mohammad Khalil, Dariya Vakhitova,
Maxim Ksenofontov, Vladislav Lamzenkov.

December 2021

1 Goal and Tasks of the Project

The goal of the project is to deploy the application consisting of two main parts (backend and frontend) Innogle on AWS via GitLab CI/CD and Docker compose.

2 Execution plan and Methodology

The App has frontend and backend parts, our job is to develop those parts, then put this project on GitLab. After finishing with coding, the essence of the project in GitLab will be the following:

- Make a *gitlab-ci.yml* file, which will have the important stages and jobs for the deployment.
- Implement the following stages in pipeline: building, testing, sending image to Docker Hub, deploying to AWS.
- Create and configure EC2 instances on AWS.
- Install and register *GitLab runner* on this EC2 instance.
- Use this GitLab Runner to run the CI/CD pipeline of *gitlab-ci.yml* file.
- Creating appropriate security groups for the instance, as a result, limiting available ports to the external world.

3 Utilization of solution and Tests

3.1 AWS Instances

We made two different EC2 instances in our private account on AWS, let's explain on details both of them:

1. The purpose of the first one was to download and register GitLab runner, because we need it to run our CI/CD pipeline. This EC2 has the following properties:
    - Amazon Linux system.
    - InstanceType: t2.micro
    - StorageSize: 20 GB
    - ImageId: eu-central-1
    - Security group presented in photo below:

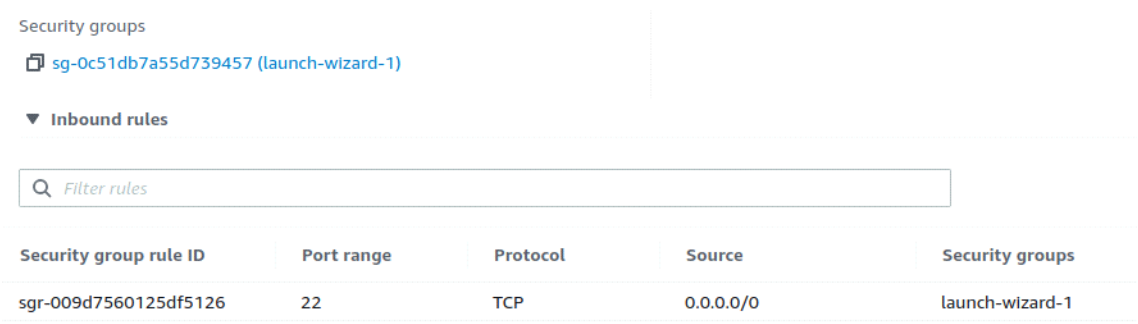


Figure 1: Security group of first AWS Instance

Then, we access AWS instances via SSH:

```
$ssh -i "mykeypair.pem"
    ec2-user@ec2-3-71-36-202.eu-central-1.compute.amazonaws.com
```

After that, we install the GitLab Runner:

```
$curl -L "https://packages.gitlab.com/install/repositories/runner/
gitlab-runner/script.rpm.sh" | sudo bash
$sudo yum -y install gitlab-runner
```

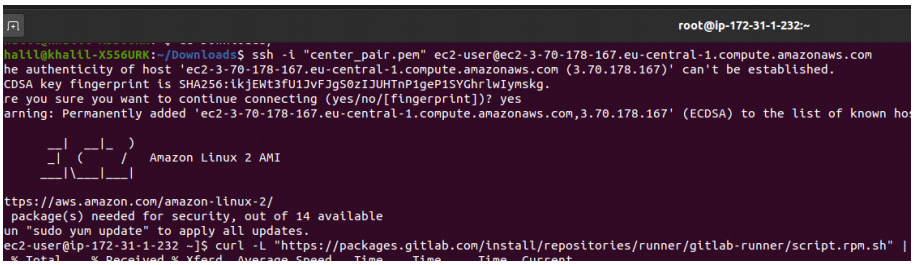


Figure 2: Establishing SSH access

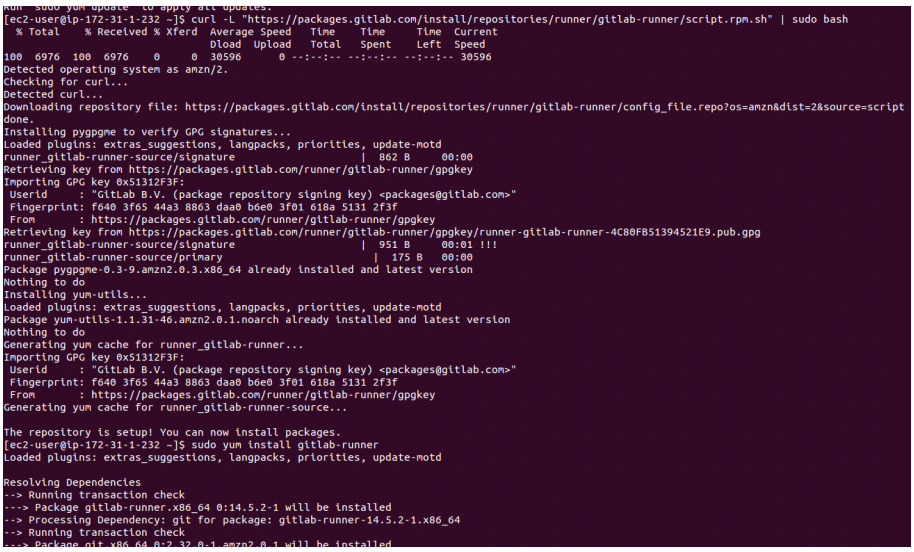


Figure 3: Installing the GitLab runner

And to register we use:

```
$sudo gitlab-runner register
```

Here is our runner's configuration from *config.toml* file:

```
concurrent = 1
check_interval = 0

[session_server]
  session_timeout = 1800

[[runners]]
  name = "innogl_runer"
```

```
url = "https://gitlab.com/"
token = "iZ3deH1QsnyLgBpfQcrx"
executor = "docker"
[runners.docker]
  tls_verify = false
  image = "docker:19.03.12"
  privileged = true
  disable_entrypoint_overwrite = false
  oom_kill_disable = false
  disable_cache = false
  volumes = ["/cache", "/certs/client"]
  shm_size = 0
[runners.custom_build_dir]
[runners.cache]
  [runners.cache.s3]
```

After that, we have our own runner!

Figure 4: GitLab Runner

2. The second AWS Instance is used for the deployment, it has almost the same properties, but with one more security group which has the following details, which make it better for accessing (TCP protocol). We open the following ports to the external world:

   - 1-8 - ports for the ICMP protocol.
   - 22 - default port for SSH protocol.
   - 80 - default port for HTTP protocol and frontend is running here.
   - 8080 - port for HTTP protocol and backend is running here.

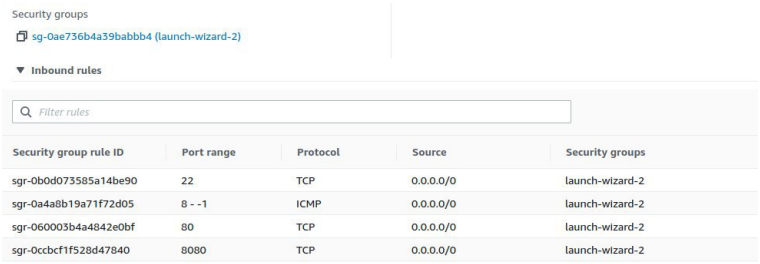


Figure 5: Security groups of second AWS Instance

3.2 Setup CI/CD pipeline

Our application is written on *Java* 11 with the use of framework *Spring Boot*. It also uses in-memory database *Redis* to store temporary user's sessions.

Now we set up the CI/CD pipeline in our GitLab repository. Our GitLab CI/CD for both the backend and frontend parts consists of 4 different stages: building, testing, sending to Docker and deploying to AWS instance.
Our file *gitlab-ci.yml*:

```
stages:
  - build
  - test
  - docker
  - deploy

include:
  - local: 'backend/ci/.gitlab-ci.yml'
  - local: 'frontend/ci/.gitlab-ci.yml'
```

As you can see from the configuration above, our main file includes a separate pipeline for backend(*backend/ci/.gitlab-ci.yml*) and frontend (*frontend /ci/.gitlab-ci.yml*).

3.2.1 General schema of pipeline

Our CI/CD pipeline can be described by the following picture:

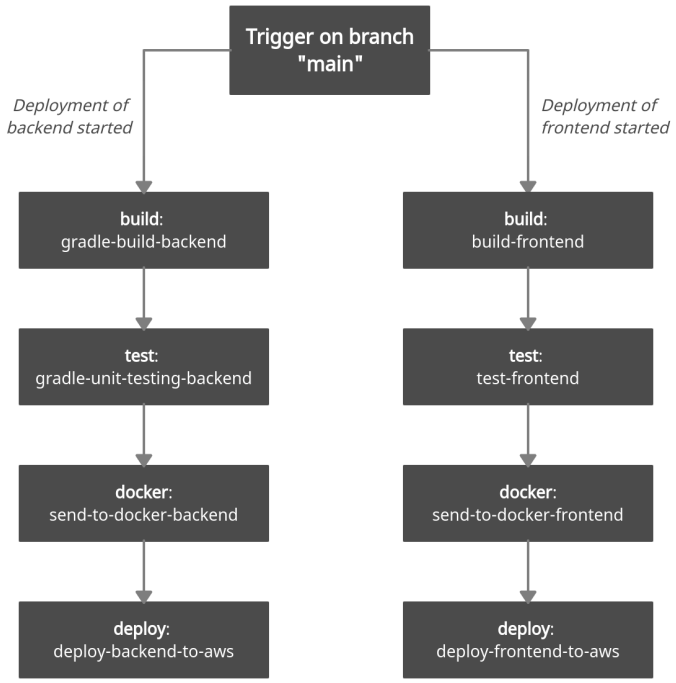


Figure 6: Example of triggering our pipeline on the "main" branch

Besides, it is important to note that the pipeline for frontend and backend are running in parallel to speed up the process of deployment of the application.

3.2.2 Pipeline for backend

Our pipeline for backend consist of 4 different stages:

- Build stage - building Spring Java project using Gradle.
- Test stage - unit testing of our Java application using Junit.
- Docker stage - sending built via docker image to our docker hub repository: https://hub.docker.com/repository/docker/mcflydesigner/innogl_backend.
- Deploy stage - deploying our backend application to the server via SSH and running it as docker container.

Our pipeline works in the following way:

- When you push commit to *main branch* the following stages are run: build, test, docker and deploy.

- When you do a merge request to *main branch* the following stages are run: test.

Our file *backend/ci/.gitlab-ci.yml*:

```
.goto-backend-project-dir: &goto-backend-project-dir
  - cd backend/

variables:
  APP_NAME: innogl_backend
  IMAGE_GRADLE: gradle:jdk11-alpine
  IMAGE_DOCKER: docker:stable
  IMAGE_UBUNTU: ubuntu:latest
  DOCKER_REPOSITORY_APP: mcflydesigner/$APP_NAME

gradle-build-backend:
  image: $IMAGE_GRADLE
  stage: build
  artifacts:
    paths:
      - backend/build/
    expire_in: 1d
  before_script:
    - *goto-backend-project-dir
    - chmod +x ./gradlew
  script:
    - ./gradlew build -x test --stacktrace
  rules:
    - if: '$CI_COMMIT_REF_NAME == "main" && $CI_PIPELINE_SOURCE !=
        "schedule"'
      when: on_success

gradle-unit-testing-backend:
  image: $IMAGE_GRADLE
  stage: test
  artifacts:
    reports:
      junit: backend/build/test-results/test/TEST-*.xml
    expire_in: 1d
  services:
    - name: docker:dind
      command: [ "--tls=false" ]
  variables:
    DOCKER_HOST: "tcp://docker:2375"
    DOCKER_TLS_CERTDIR: ""
    DOCKER_DRIVER: overlay2
  before_script:
    - *goto-backend-project-dir
  script:
    - ./gradlew test
```

```
  rules:
    - if: '$CI_MERGE_REQUEST_ID &&
        $CI_MERGE_REQUEST_TARGET_BRANCH_NAME == "main" &&
        $CI_PIPELINE_SOURCE != "schedule"'
      when: always
    - if: '$CI_COMMIT_REF_NAME == "main" && $CI_PIPELINE_SOURCE !=
        "schedule"'
      when: always

send-to-docker-backend:
  stage: docker
  image: $IMAGE_DOCKER
  variables:
    DOCKER_TLS_CERTDIR: "/certs"
  services:
    - docker:19.03.12-dind
  before_script:
    - *goto-backend-project-dir
  script:
    - docker login -p $PROD_DOCKER_PASSWORD -u $PROD_DOCKER_LOGIN
        $PROD_DOCKER_HOST
    - docker build -t $DOCKER_REPOSITORY_APP .
    - docker tag $DOCKER_REPOSITORY_APP
        $DOCKER_REPOSITORY_APP:${CI_COMMIT_SHA:0:7}
    - docker push $DOCKER_REPOSITORY_APP:${CI_COMMIT_SHA:0:7}
    - docker rmi $DOCKER_REPOSITORY_APP:${CI_COMMIT_SHA:0:7}
  rules:
    - if: '$CI_COMMIT_REF_NAME == "main" && $CI_PIPELINE_SOURCE !=
        "schedule"'
      when: on_success

deploy-backend-to-aws:
  image: $IMAGE_UBUNTU
  stage: deploy
  before_script:
    - 'which ssh-agent || ( apt-get update -y && apt-get install
        openssh-client -y )'
    - mkdir -p ~/.ssh
    - eval $(ssh-agent -s)
    - '[[ -f /.dockerenv ]] && echo -e "Host
        *\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config'
  script:
    - ssh-add <(echo "$PRIVATE_KEY")
    - ssh -o StrictHostKeyChecking=no "$SSH_USER_AND_HOST" 'sudo
        docker stop innogl-api || true; sudo docker rm innogl-api
        || true'
    - ssh -o StrictHostKeyChecking=no "$SSH_USER_AND_HOST" 'sudo
        docker stop innogl-redis || true; sudo docker rm
        innogl-redis || true'
    - ssh -o StrictHostKeyChecking=no "$SSH_USER_AND_HOST" 'rm -rf
```

```
      app-backend/ && mkdir app-backend/ && cd app-backend/ &&
      git clone https://gitlab.com/mcflydesigner/innogl.git'
  - ssh -o StrictHostKeyChecking=no "$SSH_USER_AND_HOST" 'cd
      app-backend/innogl/backend/ && docker-compose -p 8080:8080
      up -d'
  - rm -rf ~/.ssh
rules:
  - if: '$CI_COMMIT_REF_NAME == "main" && $CI_PIPELINE_SOURCE !=
      "schedule"'
    when: on_success
```

Our application is a multicontainer one, as a result, it is built and run via *docker-compose* which builds our Java application via our own *Dockerfile* (which is based on image *openjdk:15*) and image of *Redis* db.

Our file *innogl/backend/docker-compose.yml*:

```
version: "3.3"
services:
  web:
    container_name: "innogl-api"
    build: .
    ports:
      - "8080:8080"
  redis:
    container_name: "innogl-redis"
    image: "redis"
```

As you can observe in the picture above we use our own *Dockerfile* to build the Java application.

Our file *innogl/backend/Dockerfile*:

```
FROM openjdk:15
VOLUME /tmp

ENV JAR=build/libs/application-0.0.1-SNAPSHOT.jar

COPY ${JAR} app.jar

EXPOSE 8080
ENTRYPOINT ["java","-jar","/app.jar"]
```

3.2.3 Pipeline for frontend

Our pipeline for frontend consist of 4 different stages:

- Build stage - building React project using *npm*.

- Test stage - unit testing of our application.
- Docker stage - sending built via docker image to our docker hub repository: https://hub.docker.com/repository/docker/mcflydesigner/innogl_frontend.
- Deploy stage - deploying our frontend application to the server via SSH and running it as docker container.

Our pipeline works in the following way:

- When you push commit to *main branch* the following stages are run: build, test, docker and deploy.
- When you do a merge request to *main branch* the following stages are run: test.

Our file *frontend/ci/.gitlab-ci.yml*:

```
variables:
  APP_NAME: innogl_frontend
  IMAGE_NODE: node:16-alpine
  IMAGE_DOCKER: docker:stable
  IMAGE_UBUNTU: ubuntu:latest
  DOCKER_REPOSITORY_APP: mcflydesigner/$APP_NAME

.goto-frontend-project-dir: &goto-frontend-project-dir
  - cd frontend/

build-frontend:
  image: $IMAGE_NODE
  stage: build
  before_script:
    - *goto-frontend-project-dir
  script:
    - npm ci
    - CI=false
    - npm run build
  rules:
    - if: '$CI_COMMIT_REF_NAME == "main" && $CI_PIPELINE_SOURCE !=
        "schedule"'
      when: on_success

test-frontend:
  image: $IMAGE_NODE
  stage: test
  before_script:
    - *goto-frontend-project-dir
  script:
    - npm ci
    - npm test a
  rules:
```

```
    - if: '$CI_MERGE_REQUEST_ID &&
        $CI_MERGE_REQUEST_TARGET_BRANCH_NAME == "main" &&
        $CI_PIPELINE_SOURCE != "schedule"'
      when: always
    - if: '$CI_COMMIT_REF_NAME == "main" && $CI_PIPELINE_SOURCE !=
        "schedule"'
      when: always

send-to-docker-frontend:
  stage: docker
  image: $IMAGE_DOCKER
  variables:
    DOCKER_TLS_CERTDIR: "/certs"
  services:
    - docker:19.03.12-dind
  before_script:
    - *goto-frontend-project-dir
  script:
    - docker login -p $PROD_DOCKER_PASSWORD -u $PROD_DOCKER_LOGIN
        $PROD_DOCKER_HOST
    - docker build -t $DOCKER_REPOSITORY_APP --build-arg
        REACT_APP_BASE_URL=$BACKEND_GATEWAY .
    - docker tag $DOCKER_REPOSITORY_APP
        $DOCKER_REPOSITORY_APP:${CI_COMMIT_SHA:0:7}
    - docker push $DOCKER_REPOSITORY_APP:${CI_COMMIT_SHA:0:7}
    - docker rmi $DOCKER_REPOSITORY_APP:${CI_COMMIT_SHA:0:7}
  rules:
    - if: '$CI_COMMIT_REF_NAME == "main" && $CI_PIPELINE_SOURCE !=
        "schedule"'
      when: on_success

deploy-frontend-to-aws:
  image: $IMAGE_UBUNTU
  stage: deploy
  before_script:
    - 'which ssh-agent || ( apt-get update -y && apt-get install
        openssh-client -y )'
    - mkdir -p ~/.ssh
    - eval $(ssh-agent -s)
    - '[[ -f /.dockerenv ]] && echo -e "Host
        *\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config'
  script:
    - ssh-add <(echo "$PRIVATE_KEY")
    - COMMAND_TO_SERVER="sudo docker stop $APP_NAME || true; sudo
        docker rm $APP_NAME || true"
    - ssh -o StrictHostKeyChecking=no "$SSH_USER_AND_HOST"
        $COMMAND_TO_SERVER
    - COMMAND_TO_SERVER="docker run -p 3000:3000 -d --name
        $APP_NAME $DOCKER_REPOSITORY_APP:${CI_COMMIT_SHA:0:7}"
    - ssh -o StrictHostKeyChecking=no "$SSH_USER_AND_HOST"
```

```
        $COMMAND_TO_SERVER
    - rm -rf ~/.ssh
  rules:
    - if: '$CI_COMMIT_REF_NAME == "main" && $CI_PIPELINE_SOURCE !=
        "schedule"'
      when: on_success
```

Our application is built and run via *Dockerfile* which is based on image *node:16-alpine*.

Our file *innogl/frontend/Dockerfile*:

```
FROM node:16-alpine

ARG REACT_APP_BASE_URL
WORKDIR /app

ENV PATH /app/node_modules/.bin:$PATH
ENV REACT_APP_BASE_URL=$REACT_APP_BASE_URL

COPY package.json ./
COPY package-lock.json ./
RUN npm install --silent
RUN npm install react-scripts@3.4.1 -g --silent

COPY . ./

EXPOSE 3000
CMD ["npm", "start"]
```

3.3 Testing CI/CD pipeline

To verify that our CI/CD pipeline is correctly working we provide the following picture where you can see all successfully completed jobs:

- When we pushed commit $5eb0cd4a$ to *main branch* the following stages were run: build, test, docker and deploy.
- When did a merge request $4c34ef47$ to *main branch* the following stage was run: test.

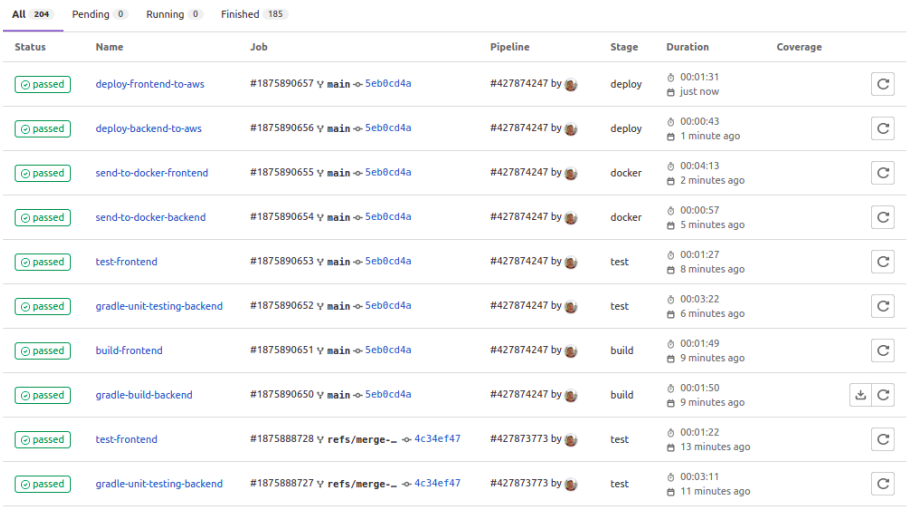


Figure 7: Successfully completed jobs in our Gitlab repository

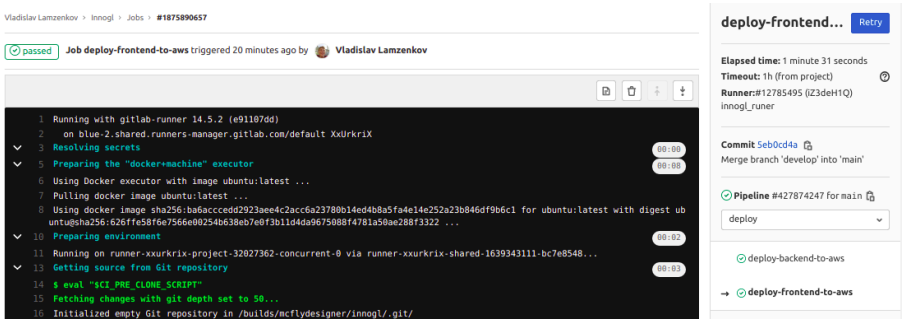


Figure 8: Our job was run on our *innogl runner*

4 Difficulties faced

4.1 Docker executor configuration

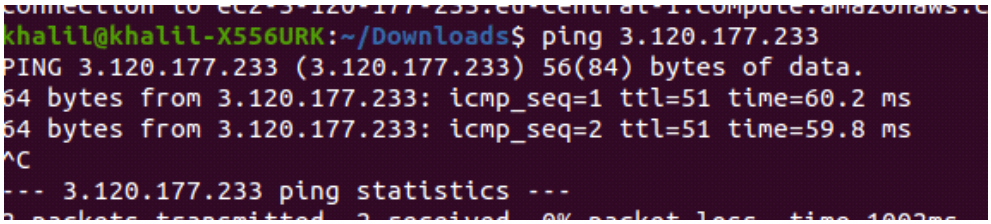
At the beginning we used *shell executor* for the gitlab runner, but then it was changed it to the *docker executor* because of its ability to work with docker directly.
That meant we had to change the configuration as well, so here is the resulting configuration to the docker executor in the *config.toml*:

```
privileged = true
disable_entrypoint_overwrite = false
oom_kill_disable = false
```

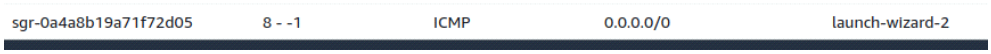
```
    disable_cache = false
    volumes = ["/cache", "/certs/client"]
    shm_size = 0
```

4.2 ICMP protocol

One more problem we faced related to the second EC2, it was that we couldn't ping the ipv4 in the beginning, but then we found out that we need to add one more security group with ICMP protocol, the problem then was solved. Here is what we had when we ping:



And here is the security group which we added:



4.3 Docker-inside-Docker

During development of the CI/CD pipeline (testing stage) we ran into the following problem: we used *testcontainers* library for our backend integration tests. In other words, running some tests implied running some docker containers. Since the tests are run in a container themselves, we basically needed to run docker-inside-docker (dind).
One more option was to use docker-out-of-docker (dood), but it was rejected due to its more complicated configuration.
Overall, we simply added a dind service to our *backend/ci/.gitlab-ci.yml* in order to make things work:

```
services:
    - name: docker:dind
```

5 Conclusion

Our team successfully implemented and deployed the application with the use of Docker Compose. During this project, we got good experience about how the full deployment of a web application (frontend-backend) is working and finish

with having a specific domain for our app which can be used anywhere.
We have been dealing with a cloud provider, CI/CD, deployment tools, software-isolation tools and security groups.
Specifically, we were working with AWS instances, GitLab-runner CI/CD pipeline, Docker tools (Docker Hub, docker-compose) and DNS tools.
It was very interesting and useful work, sometimes it was too specific, but we can't deny that we will face such problem in our industry work in the future especially considering the fact that all the team members are software developers.

6 Links

- The link to the application.
- The link to Video Demonstration.
- The link to GitLab repository