Automated Deployment Pipeline

Introduction

Continuous integration (CI) and continuous delivery (CD) are a set of operating principles and collection of practices that will enable our team to deliver changes to our application more reliably and regularly; this is known as a pipeline.

What is Continuous Integration?

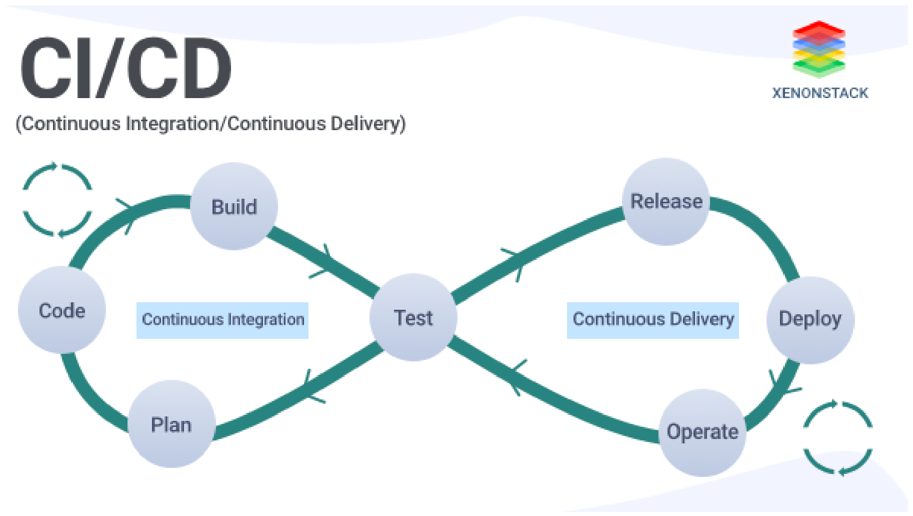
Continuous integration is an approach in which developers merge their code into a shared repository several times a day. Automated tests and software builds are run to verify the integrated code.

What is Continuous Delivery?

Continuous delivery is a strategy in which the software development team ensures that International Student Assist Application is reliable at a level high enough to release at any time. Every commit passes through the automated testing process. If it passes testing successfully, the code is considered to be ready for release into production.

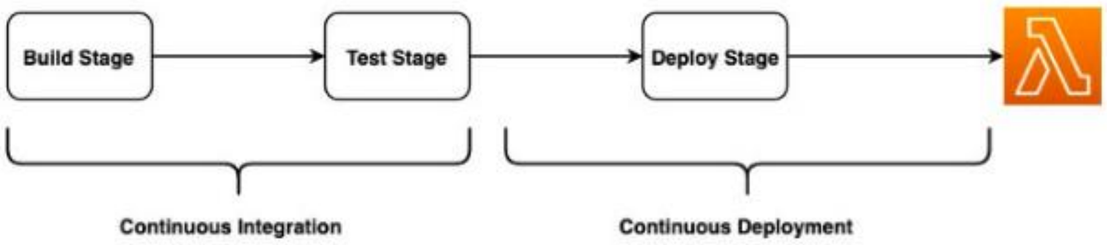
What is CI/CD Pipeline?

CI/CD stands for *Continuous Integration/Continuous Delivery*. A CI/CD pipeline is a crucial part of the modern DevOps environment. The pipeline is a deployable path that our software follows on its way to production by implementing CI/CD practices. It is a development lifecycle for any software and includes the CI/CD pipeline, which has various stages or phases through which the software passes.



CI/CD Process for International Students Assist Application

Our goal is to deploy this software application on AWS with Gitlab automatically, but we want to verify that it passes testing. The picture below illustrates the pipeline we want to build:



1. Build stage: this installs dependencies and performs type checks.
2. Test stage: this stage tests every commit.
3. Build and test Docker image: This stage builds a Docker image and publishes it.
4. Deploy stage: Deploys the Dockered Image on AWS.

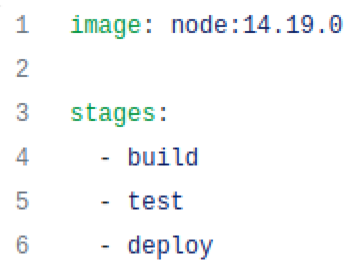
Prerequisites:

To effect our pipeline, we need the following software tools:

1. A GitLab account
2. Docker account
3. An AWS account
4. AWS CLI installed with credentials configured
5. Node.js 16.9
6. Serverless framework

Steps:

1. Create a repository: We will create a repository on GitLab with two folders called *Frontend* and *Backend* with a *ReadMe.md* file to cover both. The repository will be public and accessible to anyone. The front-end developers will work on the front-end folder; the back-end developers, on the back-end folder. Changes will be pushed to new branches, which will be tested and merged with the main branch.
2. Initialize the CI configuration file: GitLab has excellent support for CI. To enable this, we must create a hidden file called *.gitlab-ci.yml*. The file will include the following script:



```
1  image: node:14.19.0
2
3  stages:
4    - build
5    - test
6    - deploy
```

   Line 1: Tells which Docker image to be used by the Gitlab runner for the pipeline. Line 3: divides the CI process into three stages.

3. The build stage: At this stage, we will install the dependencies and perform a types check. We will update the CI configuration file to add the code below:

```yaml
install_dependencies:
  stage: build
  script:
    - npm install
    - npm run type:check
```

First, we define the job name *install_dependencies* that will run on the build stage. Then, we add scripts to be run in this job. Importantly, we can have multiple jobs in the same stage.

4. Cache the node dependencies: Installing node nodules can be time consuming, and the dependencies are installed every time we run the job. By caching these dependencies, we can speed up pipeline execution by restoring the cache in the next job. The dependencies will be reinstalled only when changes occur in the *package-lock.json* file.

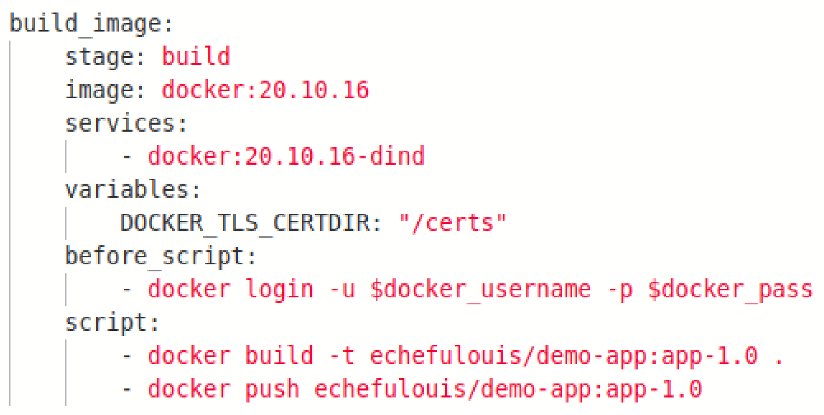
   We will define the cache in the job but at a global level since we will need these dependencies in the test and deploy stages. We can update the file *.gitlab-ci.yml* to add this snippet before the job *install_dependencies*:

```yaml
cache:
  key:
    files:
      - package-lock.json
  paths:
    - node_modules/
```

5. The test stage: We need to install dependencies and run unit tests at this stage, but we already installed the dependencies during the previous job. To avoid repetition, we will indicate that the test job depends on the build. Some tests we will run at this stage are:
   a. Async module tests.
   b. Database test.

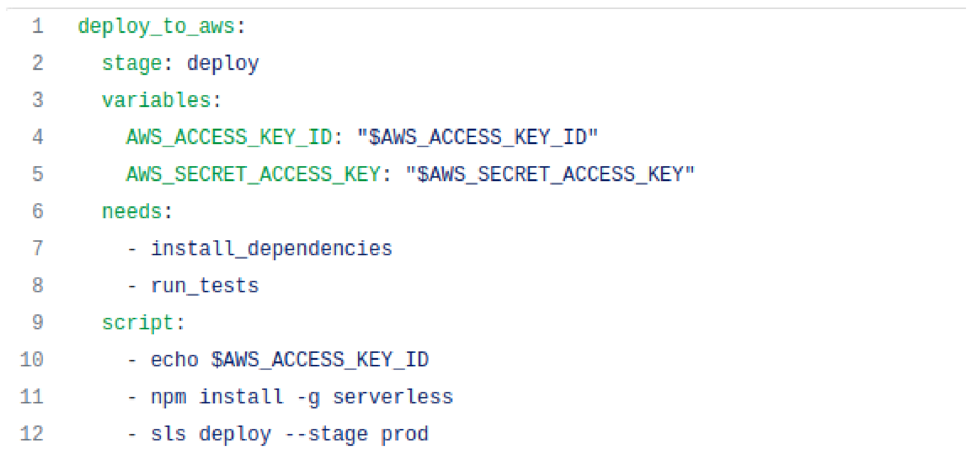
```yaml
run_tests:
  stage: test
  needs:
    - build
  script: npm run test
```

6. Build Docker image: We will use GitLab with Docker to build a Docker image for our application, which we will test and then publish to a container registry.



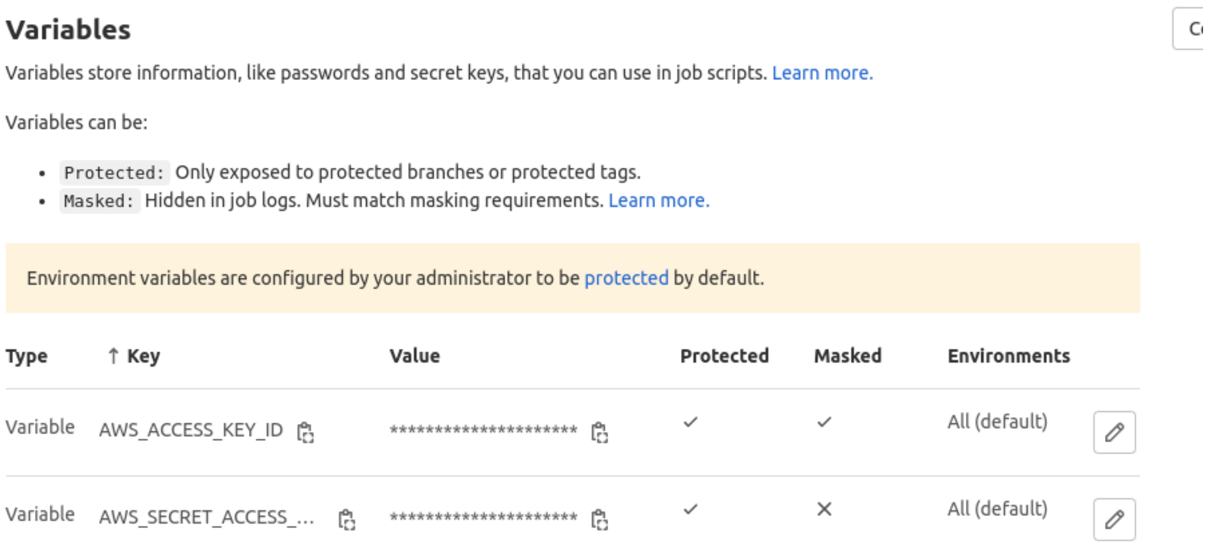
```yaml
build_image:
    stage: build
    image: docker:20.10.16
    services:
        - docker:20.10.16-dind
    variables:
        DOCKER_TLS_CERTDIR: "/certs"
    before_script:
        - docker login -u $docker_username -p $docker_pass
    script:
        - docker build -t echefulouis/demo-app:app-1.0 .
        - docker push echefulouis/demo-app:app-1.0
```

7. The deploy stage: The final stage consists of deploying the Dockered application on AWS using the Serverless Framework. Since the application is not installed, we will install it globally and then deploy it. We will update the CI configuration file and add an environment variable to the project.



```yaml
deploy_to_aws:
  stage: deploy
  variables:
    AWS_ACCESS_KEY_ID: "$AWS_ACCESS_KEY_ID"
    AWS_SECRET_ACCESS_KEY: "$AWS_SECRET_ACCESS_KEY"
  needs:
    - install_dependencies
    - run_tests
  script:
    - echo $AWS_ACCESS_KEY_ID
    - npm install -g serverless
    - sls deploy --stage prod
```

We will add two variables: AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY. These variables help store sensitive access keys, and they can be masked on Gitlab CI.


Variables

Variables store information, like passwords and secret keys, that you can use in job scripts. Learn more.

Variables can be:

- `Protected:` Only exposed to protected branches or protected tags.
- `Masked:` Hidden in job logs. Must match masking requirements. Learn more.

Environment variables are configured by your administrator to be protected by default.

Type	↑ Key	Value	Protected	Masked	Environments
Variable	AWS_ACCESS_KEY_ID	********************	✓	✓	All (default)
Variable	AWS_SECRET_ACCESS_...	********************	✓	✕	All (default)

This job depends on the two previous jobs. We do not want to reinstall the node dependencies nor to deploy the project in production if any test fails. We will commit, push, and wait for the pipeline to be completed.

What are the benefits?

1. Better code quality: we can ship fewer bugs in production because we will catch them with automated testing in the pipeline.
2. We deploy quickly: we can perform approximately 25 deployments per day.
3. Gain in productivity: our developers build a feature and do not have to spend time deploying it.
4. Rollback is easy: we can quickly revert to a previous working version when there is a critical bug (because it still happens).
5. Continuous feedback: we can deploy quickly, determine how much value is added for our customers, and iterate accordingly.