

## Continuous Deployment Pipeline with GitLab on Ubuntu

### Introduction :

Continuous Integration (CI) and Continuous Deployment (CD) are essential practices in modern software development. The primary goal of a CI/CD pipeline is to automatically build, test, and deploy your application whenever changes are made to your repository.

### Why We Need CI/CD:

A CI/CD pipeline offers significant benefits for software development and deployment:

- **Automated Deployments:** When you push changes to any branch, the CI/CD pipeline will automatically build, test, and deploy your application to your server. This reduces manual intervention and ensures consistency.
- **Faster Development Cycles:** Automation speeds up the feedback loop, allowing for quicker iterations and faster delivery of features and fixes.

### Prerequisites:

Before setting up the CI/CD pipeline, ensure your application runs via a Dockerfile. Docker allows you to package your application with all its dependencies, ensuring consistency across different environments. Please follow this for .Net application.

➤ [Documentation/Dockerizing.NET Applications.pdf at main · talukderroni13039/Documentation \(github.com\)](#)

### Step-by-Step Guide to Setting Up CI/CD Pipeline with GitLab on Ubuntu Server

#### Step 1 - DockerFile in your project

Firstly make sure your application has been containerized via Dockerfile.  
Then push your project into your specific Gitlab Repository.

#### Step 2 - Install GitLab Runner on Your Ubuntu Server

Start by logging in to your server:

```
ssh your_user_name@your_server_IP
```

Check Gitlab runner is exists or not in the server

```
systemctl status gitlab-runner
```

You will have `active (running)` in the output:

#### Output

```
● gitlab-runner.service - GitLab Runner
   Loaded: loaded (/etc/systemd/system/gitlab-runner.service; enabled; vendor preset:
   Active: active (running) since Mon 2020-06-01 09:01:49 UTC; 4s ago
 Main PID: 16653 (gitlab-runner)
    Tasks: 6 (limit: 1152)
   CGroup: /system.slice/gitlab-runner.service
           └─16653 /usr/lib/gitlab-runner/gitlab-runner run --working-directory /home
```

In order to install the `gitlab-runner` service, you'll add the official GitLab repository. Download and inspect the install script:

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

On successful execution, this returns the following message:

```
The repository is setup! You can now install packages.
```

To install the `gitlab-runner` package, run the following command in terminal:

```
sudo apt-get install gitlab-runner
```

When you execute the previous command, the output will be like:

#### Output

```
[sudo] password for sammy : % Total      % Received % Xferd  Average Speed   Time
                             Dload    Upload    Total   Spent    Left   Speed
100  5945  100  5945    0     0  8742      0  --:--:-- --:--:-- --:--:--  8729
```

### Step 3 - Register Gitlab Runner

1. In your GitLab project, navigate to **Settings > CI/CD > Runners > Expand**.
2. In the **Project runners** section, click on **New project runner** and follow the form to create a new runner for your project.
3. Once a runner is in place, you'll find the **registration token** and the **GitLab URL**. Copy both to a text editor; you'll need them for the next command. They will be referred to as `https://your_gitlab.com` and `project_token`.

Runner created.

## Register runner

GitLab Runner must be installed before you can register a runner. [How do I install GitLab Runner?](#)

### Step 1

Copy and paste the following command into your command line to register the runner.

```
$ gitlab-runner register
--url https://gitlab.com
--token
```

The **runner authentication token** displays here for a short time only. After you register the runner, this token is stored in the `config.toml` and cannot be accessed again from the UI.

### Step 2

Choose an executor when prompted by the command line. Executors run builds in different environments. [Not sure which one to select?](#)

### Step 3 (optional)

Manually verify that the runner is available to pick up jobs.

```
$ gitlab-runner run
```

This may not be needed if you manage your runner as a [system](#) or [user service](#).

[Go to runners page](#)

Back to your terminal in the server and register the runner for your project:

```
sudo gitlab-runner register -n --url https://your_gitlab.com --registration-token project_token --executor docker --description "Deployment Runner" --docker-image "docker:stable" --tag-list deployment --docker-privileged
```

[https://your\\_gitlab.com](https://your_gitlab.com)  
project\_token

Just replace two things with your gitlab runner what you have created via gitlab runner in gitlab  
Output:

### Output

Runner registered successfully. Feel free to start it, but if it's running already th

Verify the registration process by going to **Settings > CI/CD > Runners** in GitLab, where the registered runner will show up.

### Project runners

These runners are assigned to this project.

[New project runner](#)

---

### Assigned project runners

#29549129 (mSMNsVPWt)

[Remove runner](#)

[cicd](#)

### Other available runners

### Shared runners

These runners are available to all groups and projects.

Each CI/CD job runs on a separate, isolated virtual machine.

Enable shared runners for this project

☒

Available shared runners: 78

#1506020 (Hs8mheX51)

windows-shared-runners-manager-1

[shared-windows](#) [windows](#) [windows-1809](#)

## Step 4- Setting Up an SSH Key

Make sure the user has sudo access and the user is in the Docker group. This permits **deployer** to execute the **docker** command, which is required to perform the deployment.

Create ssh key pair one is public and another one is private key:

```
ssh-keygen -b 4096
```

To authorize the SSH key for the **server** user, you need to append the public key to the **authorized\_keys** file:

```
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

Start by showing the SSH private key:

```
cat ~/.ssh/id_rsa
```

Copy the output to your clipboard. Make sure to add a line break after

```
-----END RSA PRIVATE KEY-----:
```

Now navigate to **Settings > CI / CD > Variables** in your GitLab project and click **Add Variable**.

Key: **ID\_RSA**

- Value: Paste your SSH private key from your clipboard (including a line break at the end).
- Type: **File**
- Environment Scope: **All (default)**
- Protect variable: **Unchecked**
- Mask variable: **Unchecked**

Key: **SERVER\_IP**

- Value: **your\_server\_IP**
- Type: **Variable**
- Environment scope: **All (default)**
- Protect variable: **Unchecked**
- Mask variable: **Unchecked**

Key: **SERVER\_USER**

- Value: **your\_user\_name**
- Type: **Variable**
- Environment scope: **All (default)**
- Protect variable: **Unchecked**
- Mask variable: **Unchecked**

## Step 5 - Configuring the .gitlab-ci.yml File

In GitLab, go to the **Project overview** page, click the + button and select **New file**. Then set the **File name** to **.gitlab-ci.yml**.

Alternatively you can clone the repository and make all following changes to **.gitlab-ci.yml** on your local machine, then commit and push to the remote repository.

Next, add the following to your `.gitlab-ci.yml` file:

stages:

- build

variables:

```
TAG_LATEST: $CI_REGISTRY_IMAGE/$CI_COMMIT_REF_NAME:latest
TAG_COMMIT: $CI_REGISTRY_IMAGE/$CI_COMMIT_REF_NAME:$CI_COMMIT_SHORT_SHA
DOCKER_TLS_CERTDIR: ""
DOCKER_DRIVER: overlay2
```

build:

image: docker:latest

stage: build

services:

- docker:dind

only:

- deployment

tags:

- deployment

script:

```
- echo $SERVER_USER
- echo $TAG_LATEST
- echo $TAG_COMMIT
- docker build -t $TAG_COMMIT -t $TAG_LATEST .

- echo "docker push started"

- docker login -u gitlab-ci-token -p $CI_JOB_TOKEN $CI_REGISTRY
- docker push $TAG_COMMIT
- docker push $TAG_LATEST

- echo "server login started"

- echo "$ID_RSA" > /tmp/id_rsa.pem
- ls /tmp
- chmod 600 /tmp/id_rsa.pem
- cat /tmp/id_rsa.pem
- ssh -i /tmp/id_rsa.pem -o StrictHostKeyChecking=no $SERVER_USER@$SERVER_IP
- echo "login success"
- ssh -i /tmp/id_rsa.pem -o StrictHostKeyChecking=no $SERVER_USER@$SERVER_IP "docker login -u gitlab-
ci-token -p $CI_JOB_TOKEN $CI_REGISTRY"
- echo "login to the server"
- ssh -i /tmp/id_rsa.pem -o StrictHostKeyChecking=no $SERVER_USER@$SERVER_IP "docker pull
$TAG_COMMIT"
- ssh -i /tmp/id_rsa.pem -o StrictHostKeyChecking=no $SERVER_USER@$SERVER_IP "docker container rm -
f message-processor || true"
- ssh -i /tmp/id_rsa.pem -o StrictHostKeyChecking=no $SERVER_USER@$SERVER_IP "docker run -d --name
message-processor $TAG_COMMIT"
```









## Explanation:

- `docker build ...`: Builds the Docker image based on the `Dockerfile` and tags it with the latest commit tag defined in the variables section.
- `docker login ...`: Logs Docker in to the project's container registry. You use the predefined variable `$CI_JOB_TOKEN` as an authentication token. GitLab will generate the token and stay valid for the job's lifetime.
- `docker push ...`: Pushes both image tags to the container registry.
- **tags:**
  - **deployment**
- The deployment tag ensures that the job will be executed on runners that are tagged `deployment`, When create runner give them specific tag like **deployment**
- **only:**
  - **deployment**  
This is the branch name where job will be executed
- `-i` stands for **identity file** and `$ID_RSA` is the GitLab variable containing the path to the private key file.
- `-o StrictHostKeyChecking=no` makes sure to bypass the question, whether or not you trust the remote host. This question can not be answered in a non-interactive context such as the pipeline.
- `$SERVER_USER` and `$SERVER_IP` are the GitLab variables. They specify the remote host and login user for the SSH connection.
- `command` will be executed on the remote host.

## Step 6 - Validating the Deployment

When a `.gitlab-ci.yml` file is pushed to the repository, GitLab will automatically detect it and start a CI/CD pipeline. At the time you created the `.gitlab-ci.yml` file, GitLab started the first pipeline.

Go to **Build > Pipelines** in your GitLab project to see the pipeline's status. If the jobs are still running/pending, wait until they are complete. You will see a **Passed** pipeline with two green checkmarks, denoting that the publish and deploy job ran successfully.

Status	Pipeline	Created by	Stages
 Passed 🕒 00:00:53 📅 3 minutes ago	Update .gitlab-ci.yml #1074513705  main  197955c2  		 

Next click the **publish** button to open the result page of the deploy job.

## publish

✓ Passed Started 9 minutes ago by Easha

132 Cleaning up project directory and file based variables

133 Job succeeded

00:00



**Duration:** 53 seconds  
**Finished:** 9 minutes ago  
**Queued:** 0 seconds  
**Timeout:** 1h (from project) [?](#)  
**Runner:** #12270848 (ns46NMmJT) 2-green.saas-linux-small-amd64.runners-manager.gitlab.com/default

**Commit** 197955c2 [?](#)  
Update .gitlab-ci.yml

**Pipeline** #1074513705 ✓ Passed for main [?](#)

publish

Related jobs

→ ✓ publish

Finally we want to check the deployed container on our server. Head over to your terminal and make sure to log in again,

```
ssh sammy@your_server_IP
```

```
docker container ls
```

### Output

CONTAINER ID	IMAGE	COM
5b64df4b37f8	registry.your_gitlab.com / your_gitlab_user / your_project /master	

## Conclusion:

Then you configured the `.gitlab-ci.yml` pipeline configuration to:

1. Build the Docker image.
2. Push the Docker image to the container registry.
3. Log in to the server, pull the latest image, stop the current container, and start a new one.

GitLab will now deploy the application to your server for each push to the repository.

Reference:

[https://docs.gitlab.com/ee/topics/build\\_your\\_application.html](https://docs.gitlab.com/ee/topics/build_your_application.html)