Beginner Guide to Deploying an End-to-End Pipeline from GitHub to EC2 with Jenkins.

Lab Conducted by: Dalton Leyian Kimorgo

Date: 23rd November, 2023.

GitHub: https://github.com/Dalton-47

Email: daltonleyian@gmail.com

DALTON LEYIAN KIMORGO

Table of Contents

DALTON LEYIAN KIMORGO

i) Introduction:

Well first of all let us get that textbook definition for you to understand what we mean by CI/CD process.

Continuous Integration (CI) and Continuous Deployment (CD) are important practices in modern software development, revolutionizing the way code is created, tested, and delivered. CI involves developers frequently merging their code into a shared repository, triggering an automated process that compiles and tests the changes. This ensures early detection and resolution of integration issues. CD takes this a step further by automating the deployment of successful code changes to production after passing rigorous tests. The importance of automating the software delivery process is in its ability to reduce time-to-market, enhance reliability, maintain consistency across environments, and provide fast feedback to developers. By automating tasks such as testing, deployment, and version control, CI/CD pipelines contribute to efficient collaboration, scalability, and continuous improvement in software development workflows.

In this lab I will be guiding you through the process of setting up a robust Continuous Integration/Continuous Deployment (CI/CD) pipeline using Amazon EC2, Terraform, Jenkins, and GitHub. By the end of this lab, you will have a fully functional environment where code changes from your GitHub repository trigger an automated deployment to an EC2 instance.

Lab Objectives:

1. **EC2 Instance Creation with Terraform:**

Our journey begins with the creation of an EC2 instance on Amazon Web Services (AWS) using Terraform. Terraform is an Infrastructure as Code (IaC) tool that allows us to define and provision infrastructure in a declarative manner. We'll provide you with a Terraform script that automates the process of spinning up an EC2 instance.

2. **Installation of Jenkins and Apache Web Server:**

Once the EC2 instance is up and running, we'll leverage our Terraform script to install Jenkins and Apache Web Server on the instance. Jenkins is an open-source automation server, perfect for building, testing, and deploying code. Meanwhile, Apache Web Server will serve as our application host.

3. **Configuration of CI/CD Pipeline:**

With the infrastructure in place, we'll configure Jenkins to create a CI/CD pipeline. This pipeline will be connected to your GitHub repository, allowing Jenkins to automatically trigger builds and deployments whenever changes are pushed to your repository.

4. **GitHub Integration:**

DALTON LEYIAN KIMORGO

To achieve seamless automation, we'll integrate Jenkins with GitHub. This integration ensures that any updates made to your codebase trigger Jenkins jobs, initiating the CI/CD pipeline.

5. **Deployment to EC2 Instance:**

The final step involves Jenkins deploying your application to the EC2 instance. As part of the CI/CD pipeline, this deployment step ensures that the latest changes are pushed to the production environment, providing a continuous and efficient development process.

Prerequisites:

Before we dive in, make sure you have the following prerequisites in place:

- An AWS account with appropriate IAM permissions.
- Terraform installed on your local machine.
- A GitHub repository containing the code you want to deploy.

1. Installation:

1.1 Terraform installation

We are going to first install **terraform** which will be needed to automate the process of spinning up a new ec2 instance.

Head over to https://developer.hashicorp.com/terraform/install and get the right version of terraform to install in your machine, in my case I used the ubuntu/Debian Version

Ubuntu/Debian | CentOS/RHEL | Fedora | Amazon Linux | Homebrew

```
$ wget -O- https://apt.releases.hashicorp.com/gpg | sudo gpg
$ echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg
$ sudo apt update && sudo apt install terraform
```

Next ensure it is well installed on your local machine:

```
└$ terraform -version
Terraform v1.6.3-dev
on linux_amd64
```

1.2 Check AWS CLI Version

```
└$ aws --version
aws-cli/2.13.7 Python/3.11.4 Linux/6.1.0-kali9-amd64 exe/x86_64.kali.2023 prompt/off
```

If you do not have aws for cli installed head over to https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html and follow the procedure to install the right version for your local machine.

I have installed for Kali Linux OS.

1.3 Write Terraform Configuration

Create a Terraform script (e.g., `main.tf`) defining AWS resources (web server, etc.

I will provide you with a basic script needed for beginner setup

- The script launches an EC2 instance with a specified Amazon Machine Image (AMI), instance type (t2. micro), and user data script.
- The AMI used will be "ami-0c55b159cbfafe1f0," which is the Amazon Linux 2 AMI.
- The `user_data` block contains a bash script that runs on the newly created EC2 instance.
- The script updates the system, installs Apache HTTP Server, Java (OpenJDK 11), and Jenkins.
- It starts and enables the services for HTTP Server and Jenkins.
- Finally, it creates a simple "Hello, World!" HTML page at `/var/www/html/index.html`.

1.4 Creating directory for your project

Create a directory for your Terraform project. You can name it whatever you like, for example, `terraform-aws-webserver`

```
└$ mkdir terraform-aws-webserver
```

```
└$ cd terraform-aws-webserver
```

1.5 Create Terraform Script:

Create a file named `main.tf` using a text editor of your choice. This file will contain the Terraform configuration.

```
└$ touch main.tf
```

DALTON LEYIAN KIMORGO

1.6 Edit Terraform Script:

Open `main.tf` in a text editor and define your AWS resources.

To get a valid AMI Id for your Amazon Linux 2 in the US-East-1 region, you can use the following command:

```
└─$ aws ec2 describe-images --owners amazon --filters "Name=name,Values=amzn2-ami-hvm-*-x86_64-gp2" --query 'Images | [0].ImageId'
```

This gave me this id:

```
"ami-0806bc468ce3a22ec"
```

Now here is the script:

```
# AWS Provider Configuration
provider "aws" {
  region = "us-east-1" # Change this to your desired AWS region
}

# DEFAULT VPC
resource "aws_default_vpc" "default_vpc" {}

# SECURITY GROUP
resource "aws_security_group" "security_group" {
  name   = "security_group"
  vpc_id = aws_default_vpc.default_vpc.id

  # SSH INGRESS
  ingress {
    from_port   = "22"
    to_port     = "22"
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  # HTTP
  ingress {
    from_port   = "80"
    to_port     = "80"
```

DALTON LEYIAN KIMORGO

```
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  #Custom TCP for Jenkins
  ingress {
    from_port = "8080"
    to_port   = "8080"
    protocol  = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  # HTTPS
  ingress {
    from_port   = "443"
    to_port     = "443"
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]

  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = -1
```

```
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    name = "security_group"
  }
}

# EC2 INSTANCE
resource "aws_instance" "webserver" {
  ami           = "ami-0806bc468ce3a22ec"  # Amazon Linux 2 AMI ID
  instance_type = "t2.micro"

  vpc_security_group_ids = [aws_security_group.security_group.id]  # Attach the security group

  tags = {
    Name = "webserver-instance"
  }
```

```
  user_data = <<-EOF
              #!/bin/bash
              sudo yum update -y
              sudo yum install -y httpd
              systemctl start httpd
              systemctl enable httpd

              # Install Java and Jenkins
              amazon-linux-extras install -y java-openjdk11
              sudo wget -O /etc/yum.repos.d/jenkins.repo http://pkg.jenkins-ci.org/redhat-stable-rc/jenkins.repo
              sudo rpm --import http://pkg.jenkins-ci.org/redhat-stable-rc/jenkins-ci.org.key
              sudo yum clean all
              sudo yum install -y jenkins

              systemctl start jenkins
              systemctl enable jenkins

              echo "<h1>Hello, World!</h1>" > /var/www/html/index.html
              EOF
}

output "webserver-Public-URL" {
  value = aws_instance.webserver.public_ip
}
```

1.7 Initialize Terraform:

Run the following command to initialize Terraform in your project directory:

```
└─$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v5.26.0...
- Installed hashicorp/aws v5.26.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

It takes a couple of minutes to install the required plugins, be patient.

1.8 Apply Terraform Configuration:

Apply the Terraform configuration to create the resources. During this process, Terraform will provide an output that includes the `webserver-Public-URL`.

```
└─$ terraform apply
```

On success you will get this (your IP will be different):

```
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

webserver-Public-URL = "3.88.201.50"
```

Open a web browser and type in the public IP, you should be seeing this:

Hello, World!

Now head over to your AWS dashboard, you should see the instance running:

Running t2.micro 2/2 checks passed View alarms +

2.0 Configure Jenkins

Select your EC2 instance on the dashboard and click connect.

```
$ systemctl status jenkins
```

I got this showing it is inactive:

```
jenkins.service - Jenkins Continuous Integration Server
 Loaded: loaded (/usr/lib/systemd/system/jenkins.service; disabled; vendor preset: disabled)
 Active: inactive (dead)
```

Start Jenkins then enable it to make it active:

```
sudo systemctl start jenkins
```

```
sudo systemctl enable jenkins
```

Install Git:

```
sudo yum install git
```

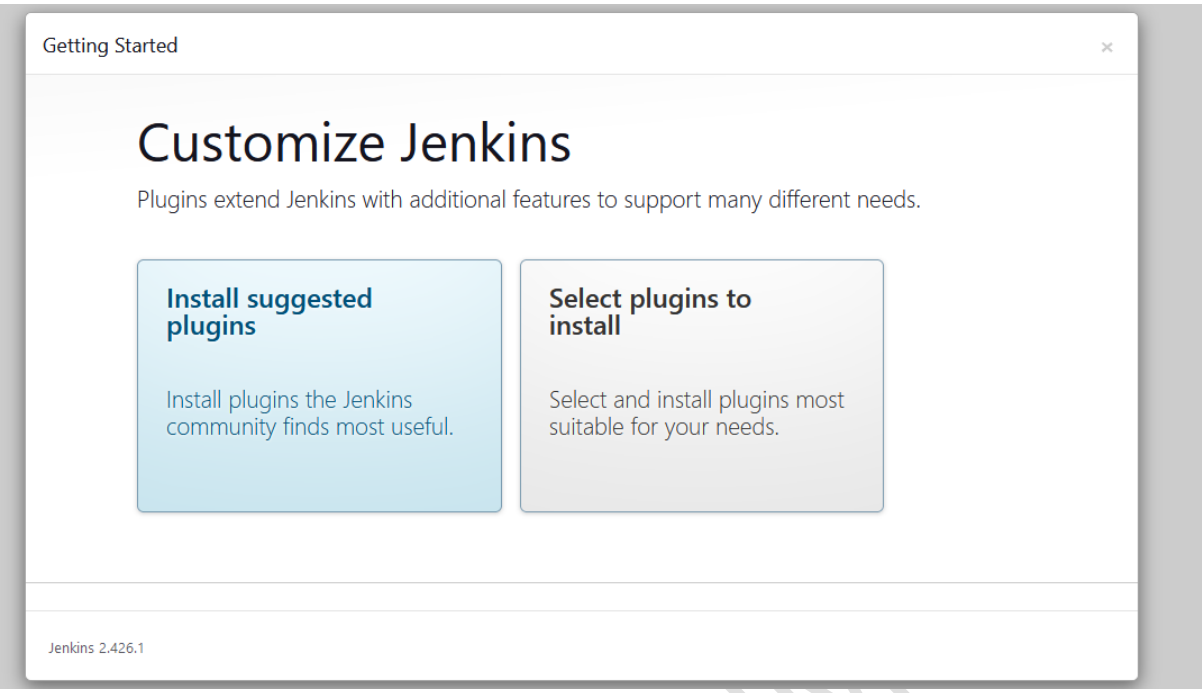
Run the public IP of your ec2 instance with port 8080, in my case I will open **3.88.201.50:8080**

You should be getting the screen above!

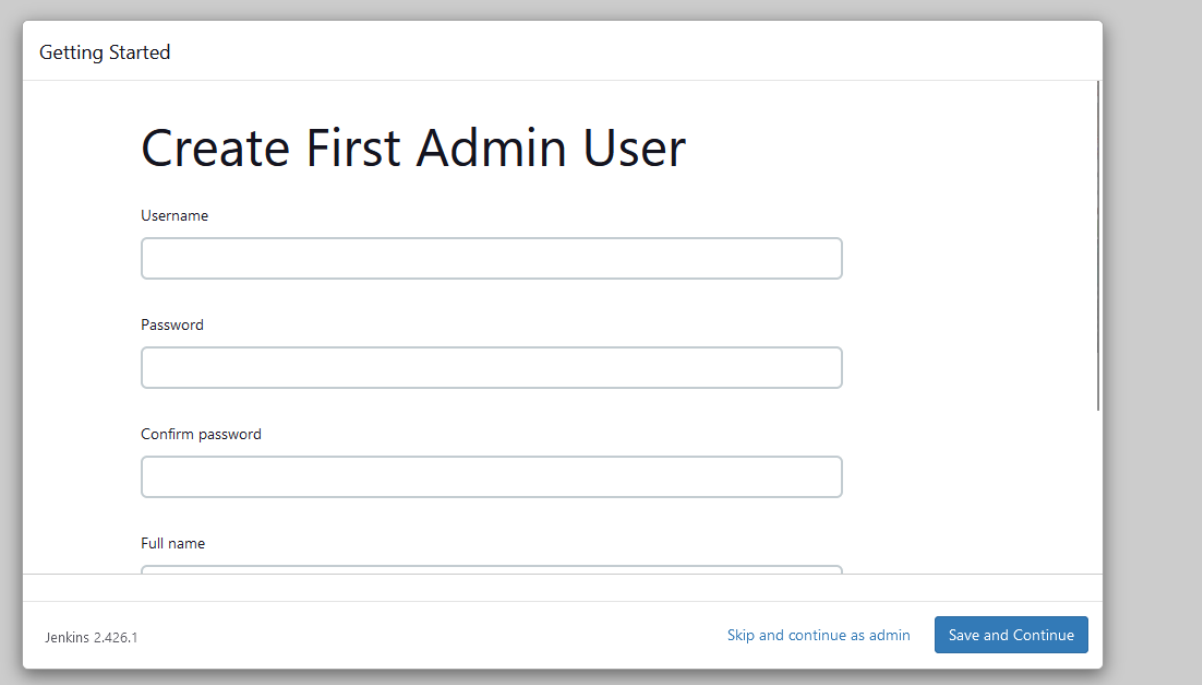
Copy the path provided above then on your ec2 open the file in the path with the cat command i.e

```
sudo cat /var/lib/jenkins/secrets/initialAdminPassword
```

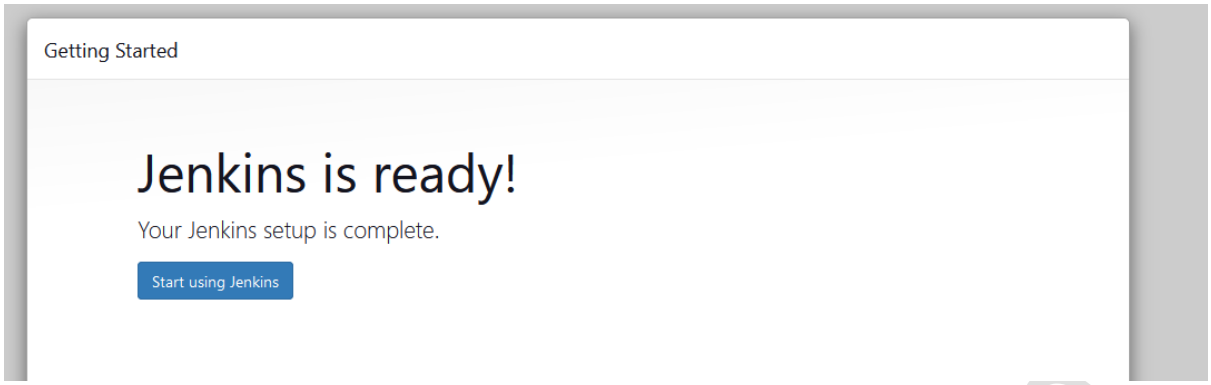
Next page you should select the option to Install Suggested Plugins.



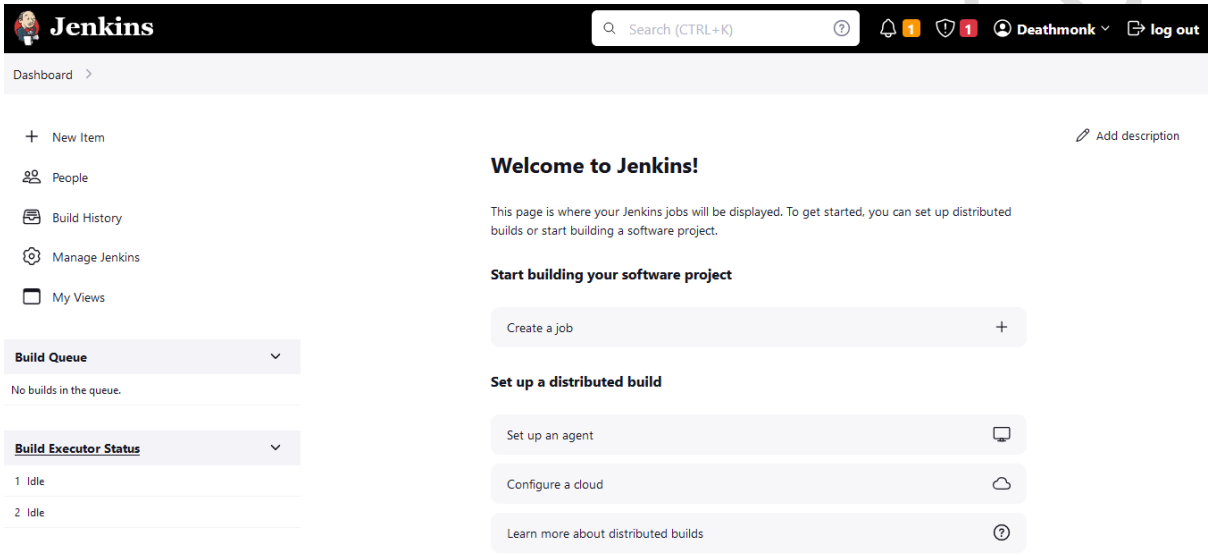
Input your credentials, ensure to remember username and password since you will be using it to login later.



Click save and continue, and in the next screen don't change the Jenkins URL configuration, click save and finish.

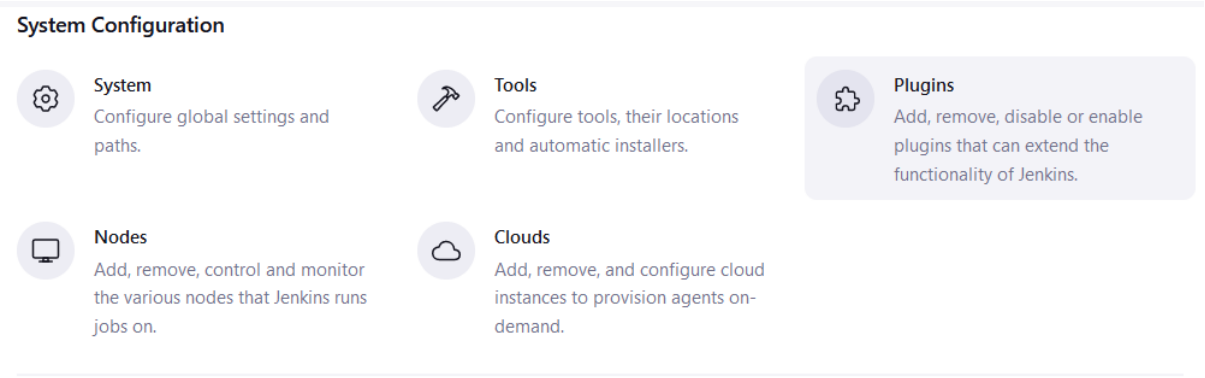


Here we go!!

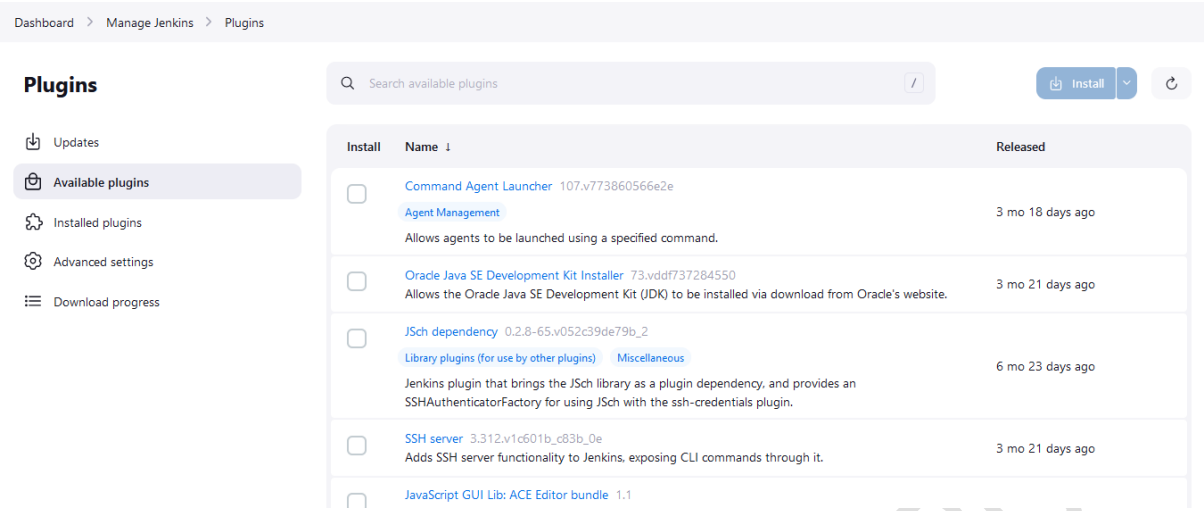


First of all we need to install all Github and Git plugins since we will be deploying a website from a github repo, so headover to manage Jenkins on the options provided on the left.

2.1 Select Plugins



Select Available plugins



Scroll the available plugins while selecting these plugins for installation:

Git Server

Github Integration

Github Authentication

Pipeline: Github

REPO

Github Checks

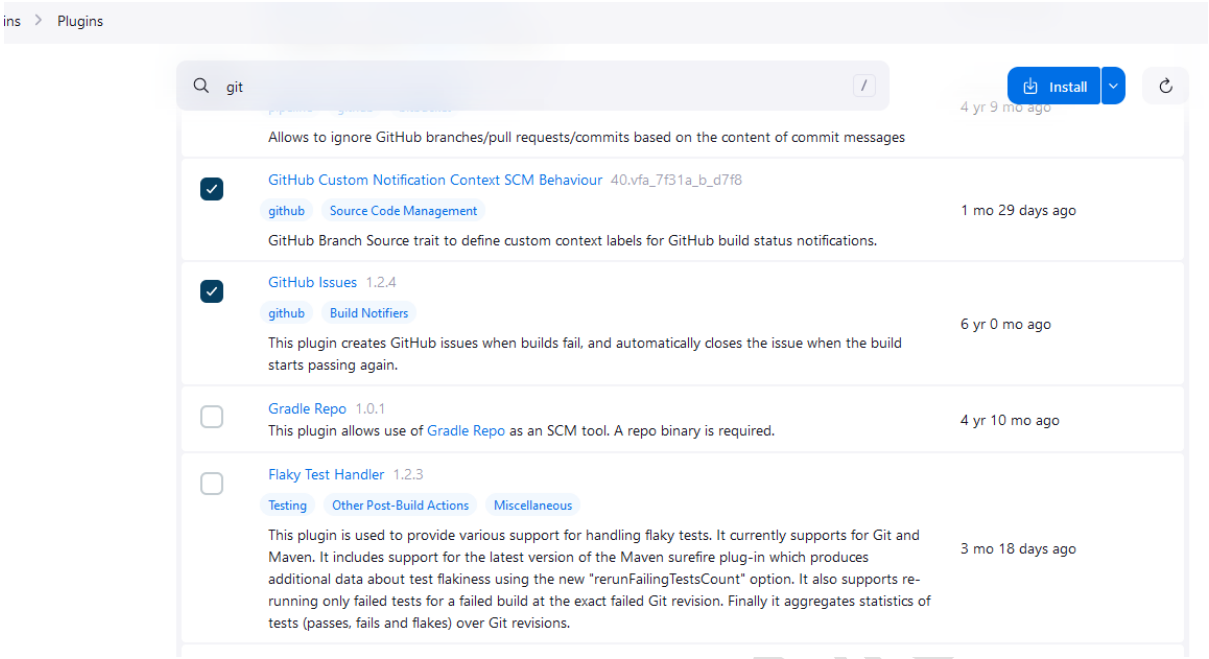
Git Push

Github Commit Skip SCM Behaviour

Github Issues

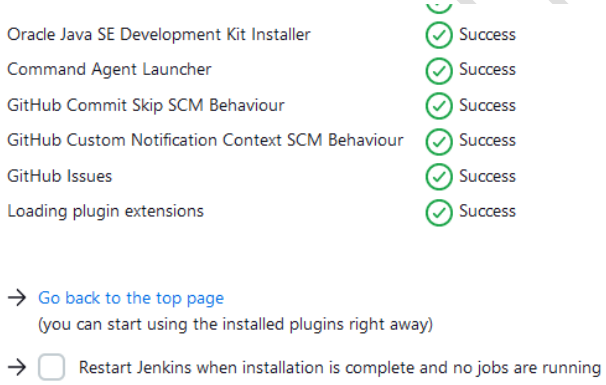
Publish Over SSH

DALTON LEYIAN KIMORGO

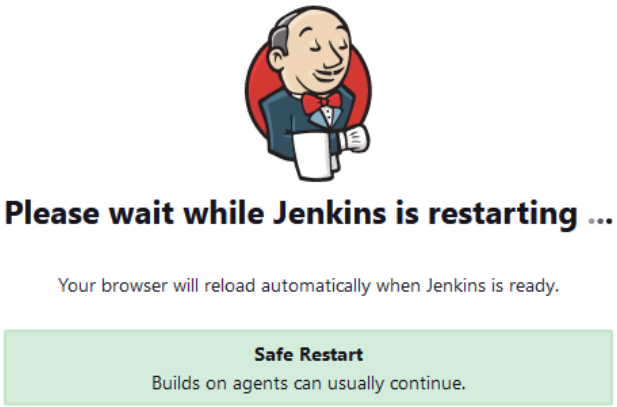


Click Install after selecting the relevant plugins.

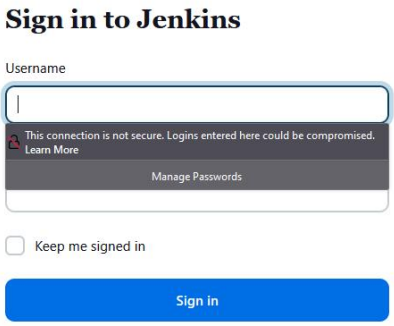
After Installation check the option to Restart Jenkins when installation is complete….



Wait for the restart

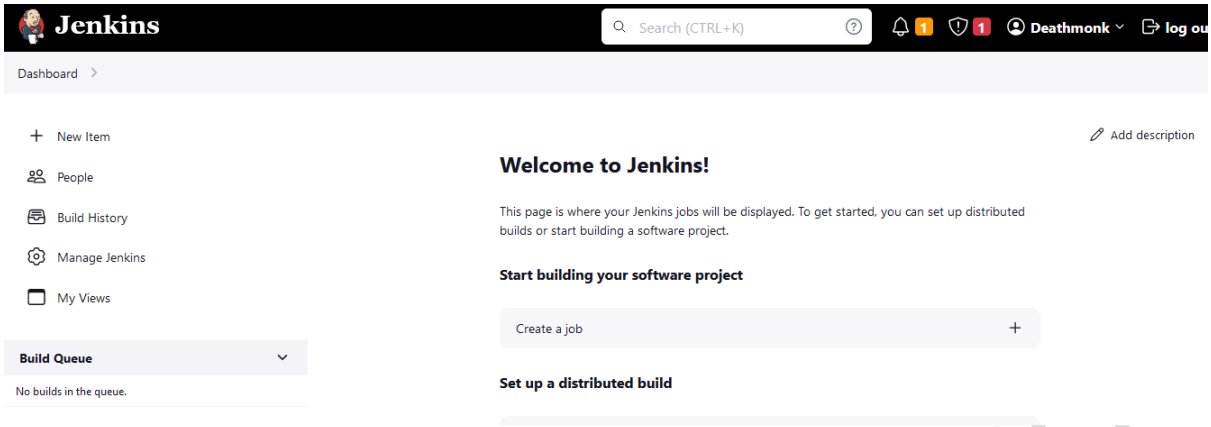


You will be prompted to sign in

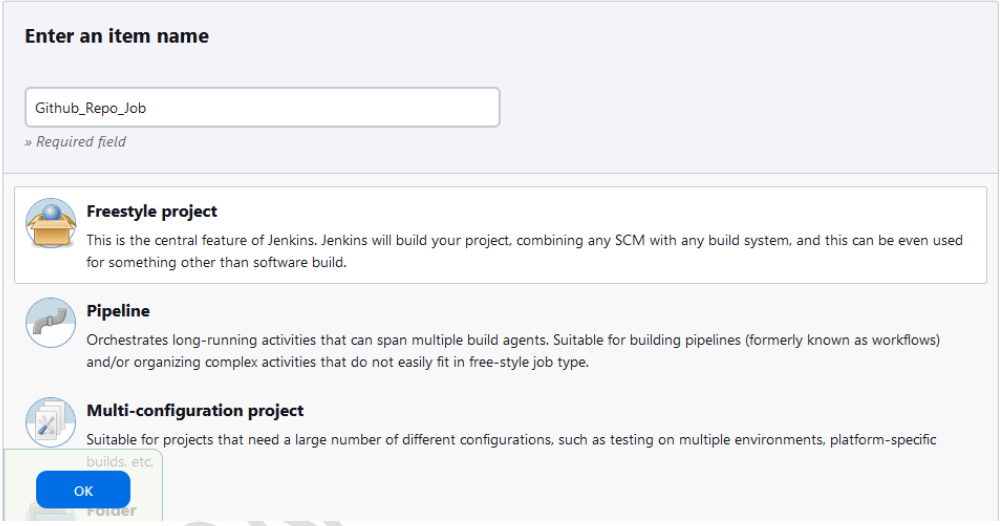


Let us build a job to listen to any push-commits on our repo.

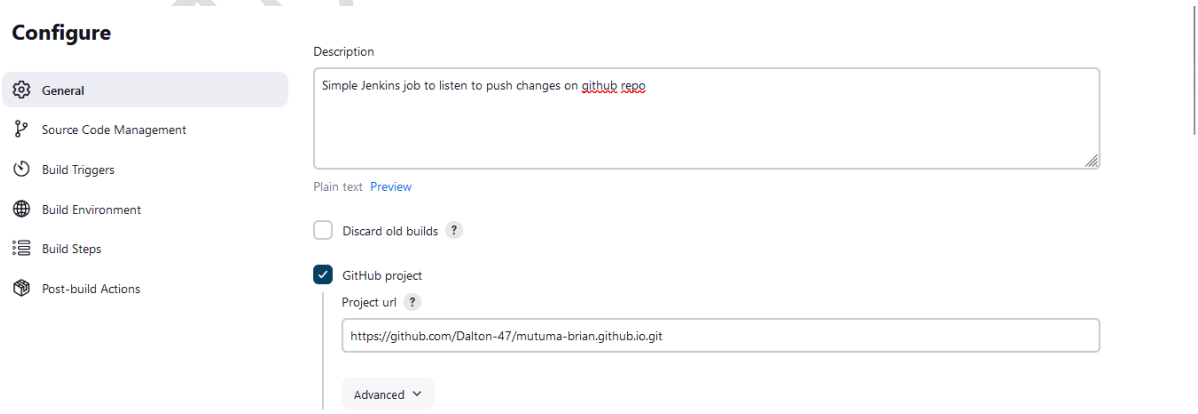
Select New Item on the Github DashBoard



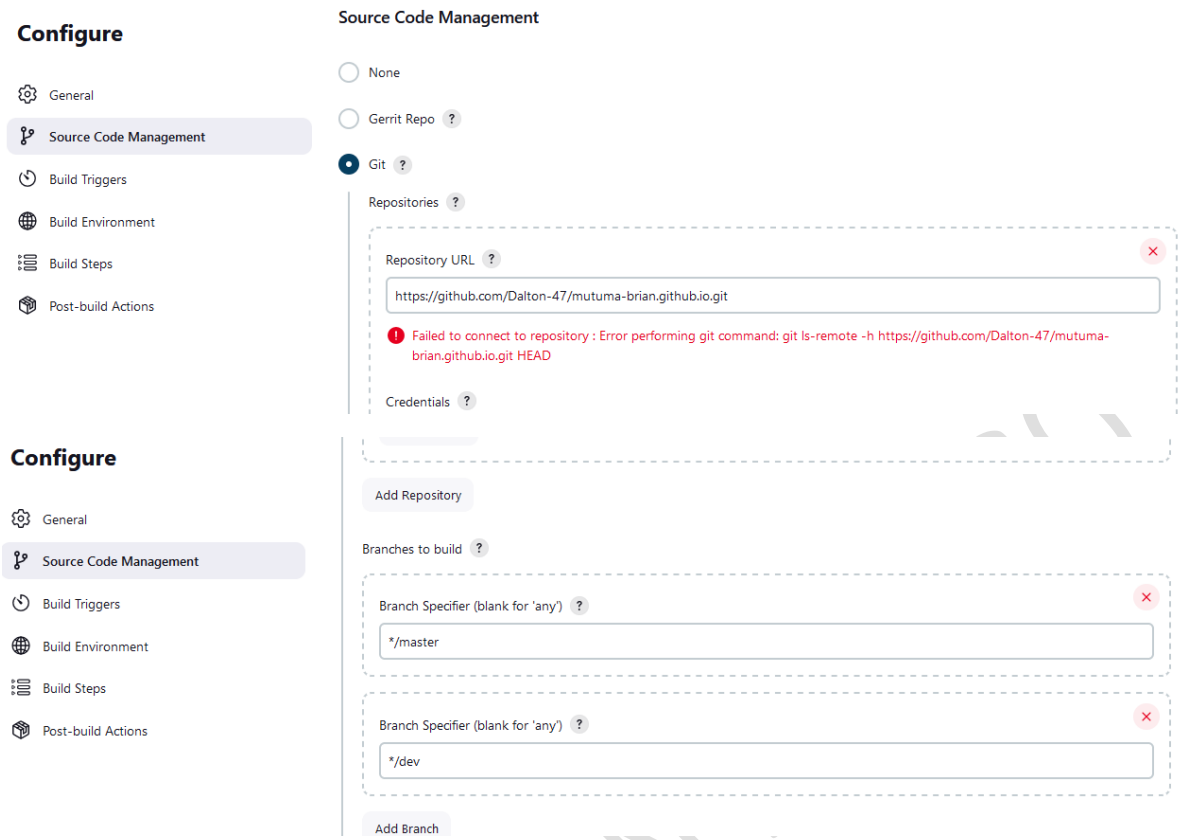
Next Step you will be required to enter an Item Name then select The first option i.e Freestyle Project then click okay



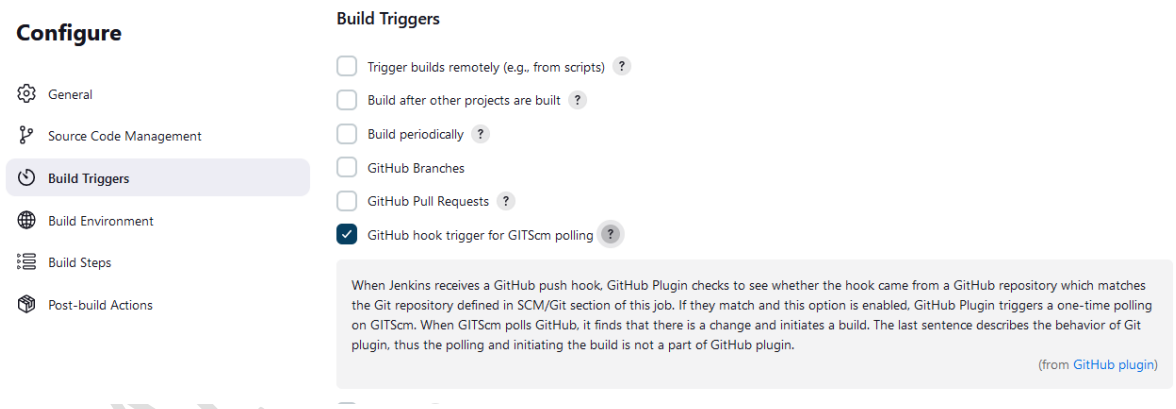
Select Github Project then pass link to the repo you want changes listened to:



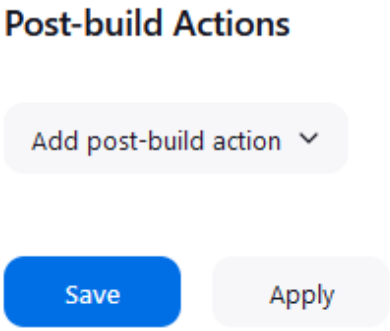
On Source Code Management select Git and pass the repo url, you can specify the branch to listen on too, default branch is master, add any other you may want.



On Build Triggers select Github hook trigger for GITScm polling



Save the changes



2.2 Adding Github WebHooks

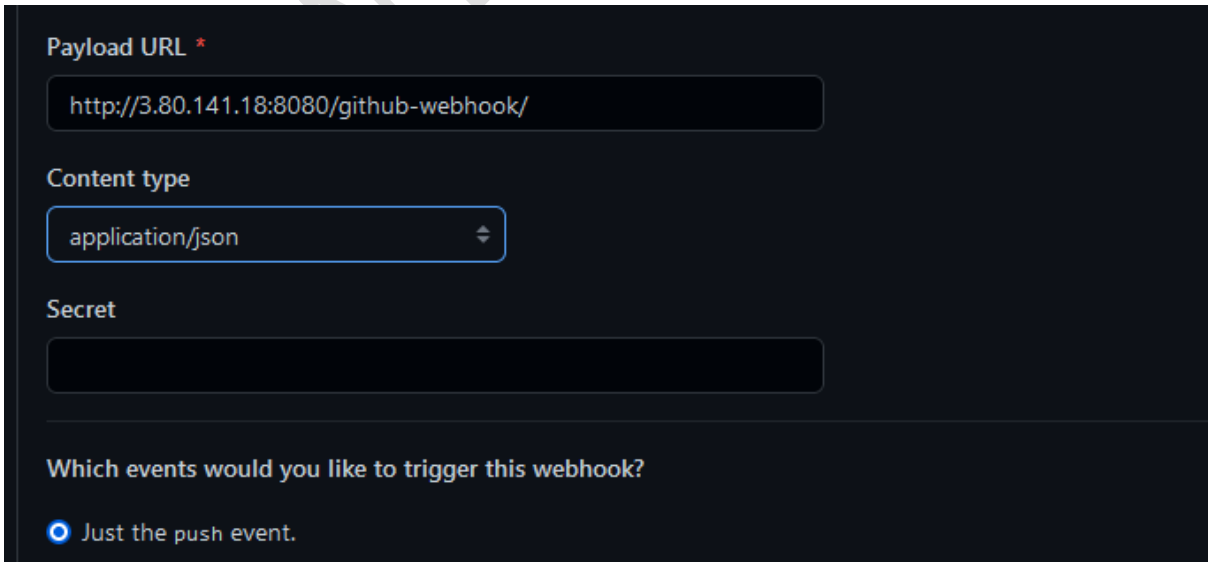
I will be adding GitHub Web Hooks for triggering Jenkins jobs when changes are pushed to your GitHub repository.

- Open your GitHub repository in a web browser.
- Navigate to the repository settings.
- Click on "Webhooks" in the left sidebar.
- Click on "Add webhook."

In the "Payload URL" field, enter the Jenkins GitHub webhook URL. It should be in the format: `http://your-jenkins-server/github-webhook/`

I will leave the secret part blank since I did not state any credentials on my Jenkins job.

Select the push events then save the webhook.



Back to Jenkins and click on the build now for our job:

You should have a successful Build.

Console Output

```
[Checks API] No suitable checks publisher found.
Finished: SUCCESS
```

S	W	Name ↓	Last Success	Last Failure	Last Duration
✓		Github_Repo_Job	1 min 58 sec #3	5 min 53 sec #2	7.2 sec

Next step we will create a job that initiates a remote execution script on our server which updates the local repository on the ec2 instance and automatically push the changes to be deployed.

First we need to generate ssh key for connecting to our ec2 instance when executing the remote script:

Let us ensure that we don't have any existing ssh key on our ec2 AMI :

```
cd ~/.ssh
```

```
ls -al
```

If the output from the above looks like this then we will generate new key.

```
drwx------ 2 ec2-user ec2-user  29 Nov 23 11:06 .
drwx------ 5 ec2-user ec2-user 146 Nov 23 14:22 ..
-rw------- 1 ec2-user ec2-user   0 Nov 23 11:06 authorized_keys
```

Since I have no keys I will generate new keys and copy them to the authorized keys,

```
ssh-keygen -t rsa -b 2048 -f ~/.ssh/id_rsa
```

You will then be prompted to enter a passphrase, ensure to note it down or remember it as we will use it for our ssh connections.

The output below indicates that our keys have been generated successfully i.e both public & private keys.

```
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/ec2-user/.ssh/id_rsa.
Your public key has been saved in /home/ec2-user/.ssh/id_rsa.pub.
```

DALTON LEYIAN KIMORGO

Next step we will copy our public key to the set of authorized keys so that it can be recognized during ssh connections. Also we will set the required permissions to read into the files.

```
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
chmod 700 ~/.ssh
chmod 600 ~/.ssh/authorized_keys
```

Try to ssh into your ec2 instance and check if the process was a success:

Replace the your-ec2-instance-ip with a valid public IP of your ec2 instance.

```
ssh -i ~/.ssh/id_rsa ec2-user@your-ec2-instance-ip
```

If the ssh login was a success you will see the output below :

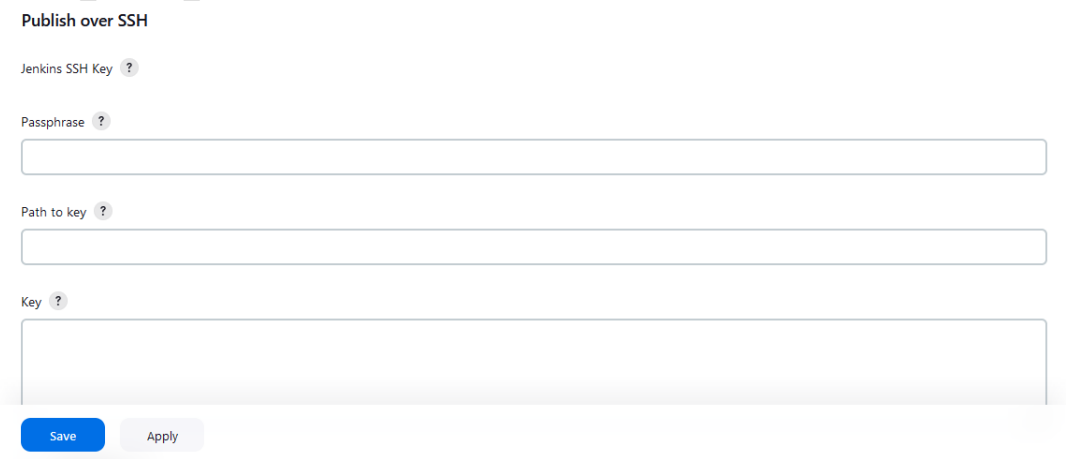
```
 ,     #_
 ~\_  ####_        Amazon Linux 2
~~  \_#####\
~~     \###|       AL2 End of Life is 2025-06-30.
~~       \#/ ___
 ~~       V~' '->
  ~~~         /    A newer version of Amazon Linux is available!
    ~~._.   _/
       _/ _/       Amazon Linux 2023, GA and supported until 2028-03-15.
     _/m/'           https://aws.amazon.com/linux/amazon-linux-2023/
```

2.3 Remote Script Configuration

Next step we will be Configuring our connection over SSH and ensuring it is a success by testing

On the Jenkins dashboard click on Manage Jenkins > System > Scroll down and locate Publish Over SSH.

Note: This will only appear if you installed the Publish Over SSH plugin.



Click add server then configure it accordingly:

Name = Name of my ec2 Instance

Hostname = where my site will be hosted which is same as my ec2 instance public IP

Username = username to connect to my ec2 instance which is ec2-user in my case,

Click advanced option then set the passphrase and private key for connection

To get your private key use the cat command which will be:

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

Copy the key contents from

-----BEGIN RSA PRIVATE KEY-----

Everything here

To

-----END RSA PRIVATE KEY-----

Paste the above key to the key textbox as shown below

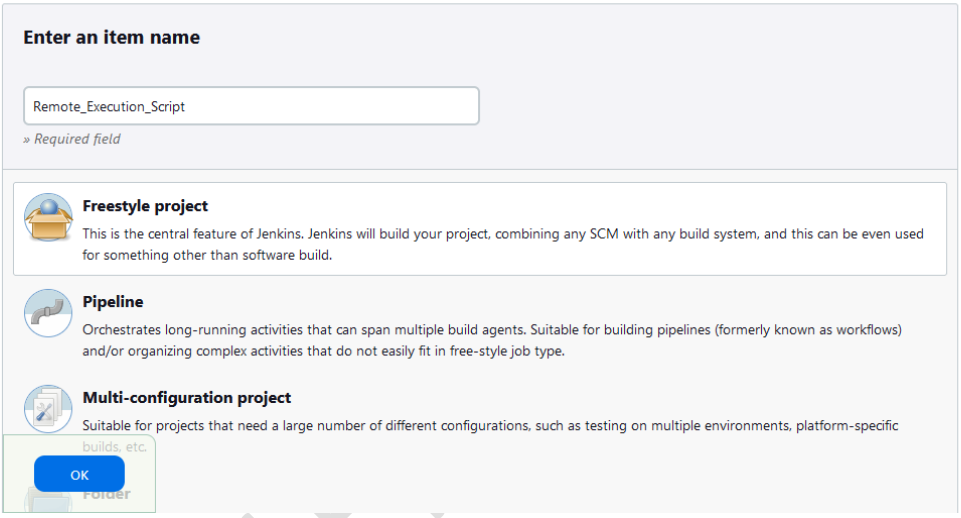
DALTON LEYIAN KIMORGO

Scroll down and click on test configuration:

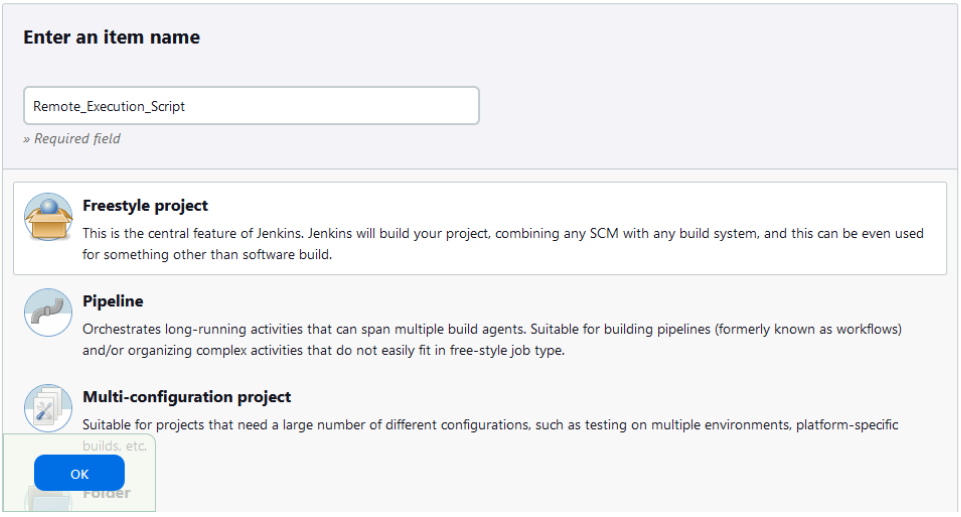
This should be your outcome:

Save the changes.

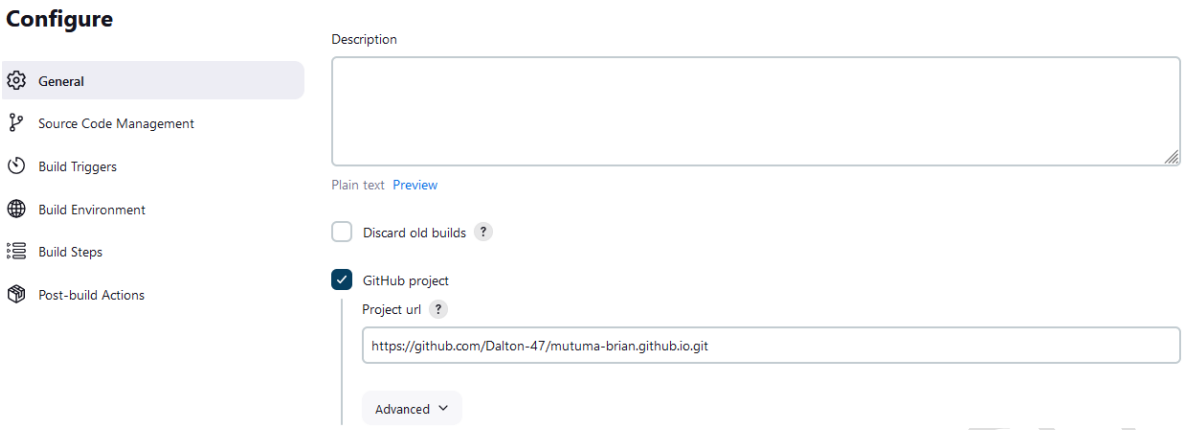
Now we create the remote execution job:



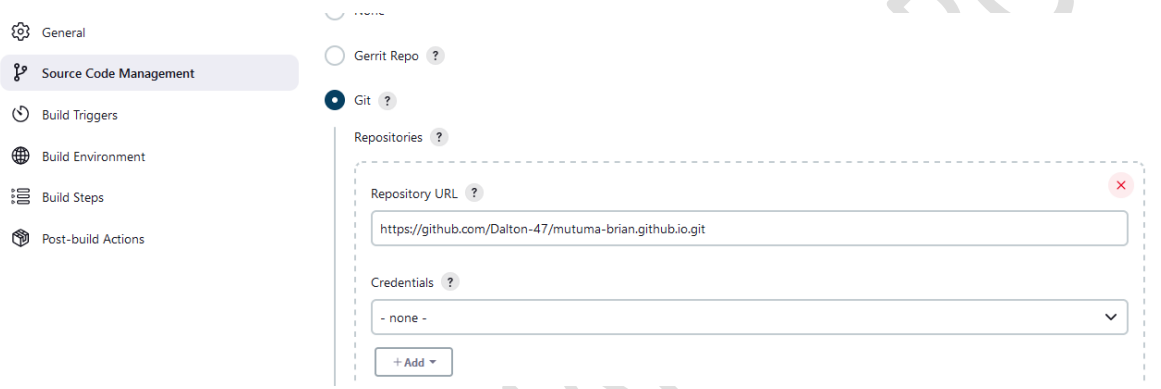
First of all we need to create a new job so select new Item and name it, select freestyle project and click ok:



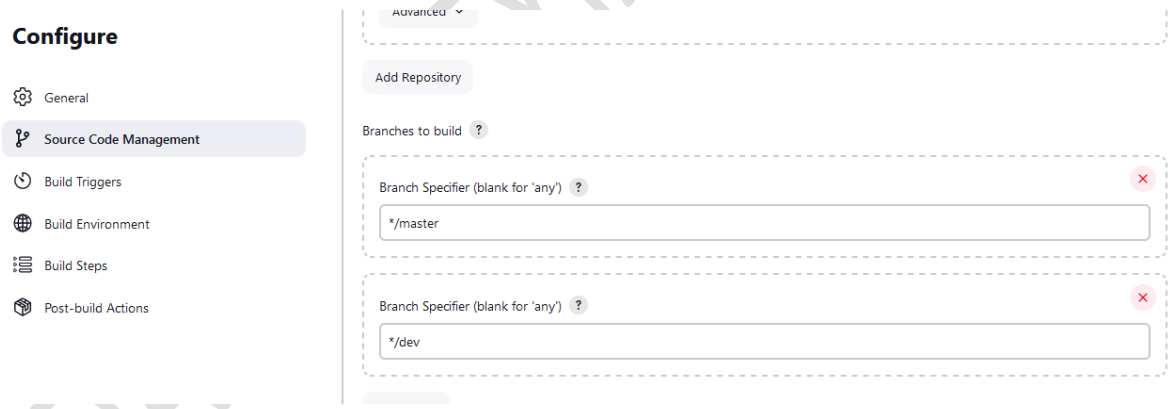
Select Github project and pass the link to your repo:



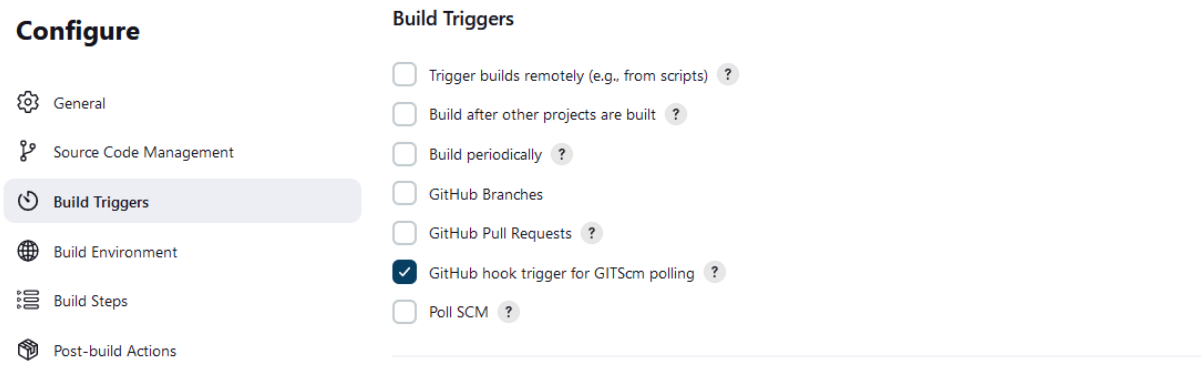
On Source Code Management (SCM) paste the link to your repo in the Git Repo URL



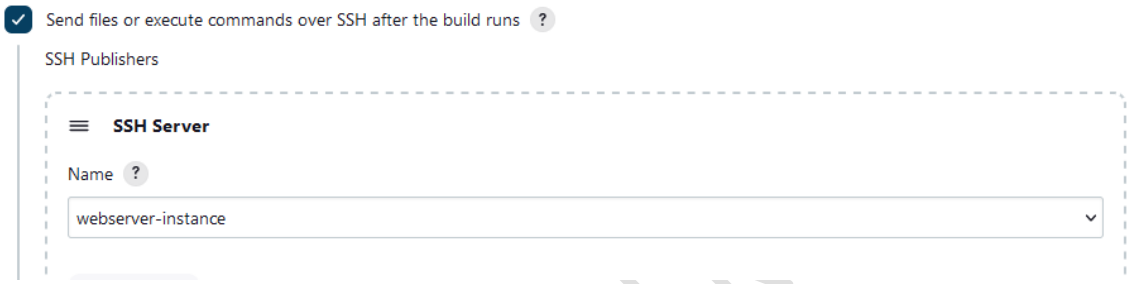
Set your branch



On Build Triggers Select **Github Hook trigger for GITScm polling** which basically sets Jenkins to listen when a GitHub push hook is received, GitHub Plugin checks to see whether the hook came from a GitHub repository which matches the Git repository defined in SCM/Git section of this job. If they match and this option is enabled, GitHub Plugin triggers a one-time polling on GITScm. When GITScm polls GitHub, it finds that there is a change and initiates a build. The last sentence describes the behavior of Git plugin, thus the polling and initiating the build is not a part of GitHub plugin.

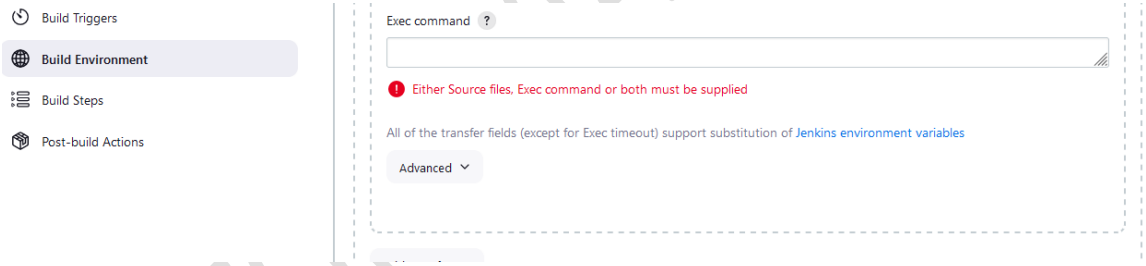


On the Build Environment select the option to send files or execute commands over SSH after the build runs.



Since we already pre-configured our ssh server we will just select it as shown above.

Scroll down to the Exec Command still in Build environment



Now we need to write an executable shell script on our ec2 instance and save it as deploy.sh and make it executable.

First of all clone your repo to your ec2 instance:

```
git clone https://github.com/Dalton-47/mutuma-brian.github.io.git
```

Next step you create the executable shell:

```
nano deploy.sh
```

Write the script to be executed, in my case I want the script to pull changes from GitHub repo and push any changes from local repository just in case of a merge conflict then copy the file changes to root directory to be deployed.

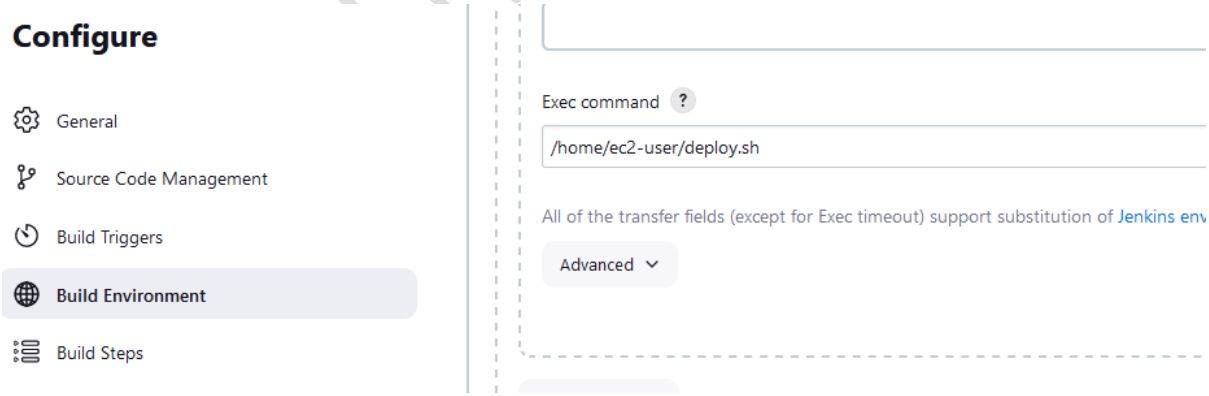
```
#!/bin/bash

# Path to your local GitHub repository
local_repo_path="/home/ec2-user/mutuma-brian.github.io"

# Path to your private key
private_key_path="/home/ec2-user/.ssh/id_rsa"

# SSH username and server address
ssh_user="ec2-user"
ssh_server="3.80.141.17"

# Path to the web server's document root
remote_document_root="/var/www/html"

# Navigate to the local GitHub repository
cd "$local_repo_path"
```

```
# Pull the latest changes from GitHub
git pull
git add .
git commit -m "done updating"

# Copy files to the web server's document root
sudo rsync -avz --no-o --no-g -e "ssh -i $private_key_path" "$local_repo_path/" $ssh_user@$ssh_server:$remote_document_root

# Restart Apache (adjust the command based on your web server)
sudo service httpd restart
```

We have to make the script executable:

```
chmod +x deploy.sh
```

Finally in our execution command under Build Environment we pass the path to our executable script shell and apply the changes then save.



Lastly ensure that your directory /var/ww/html has appropriate read and write permissions like this:

drwxrwxr-x 2 ec2-user ec2-user 4096 Nov 23 18:22 /var/www/html

You can use thes commands :

```
sudo chmod 775 /var/www/html
```

```
sudo chown -R apache:apache /var/www/html
```

```
sudo chown -R ec2-user:ec2-user /var/www/html
```

Ensure also your private key file has the right permissions i.e., it should only be readable by the user running the script command.

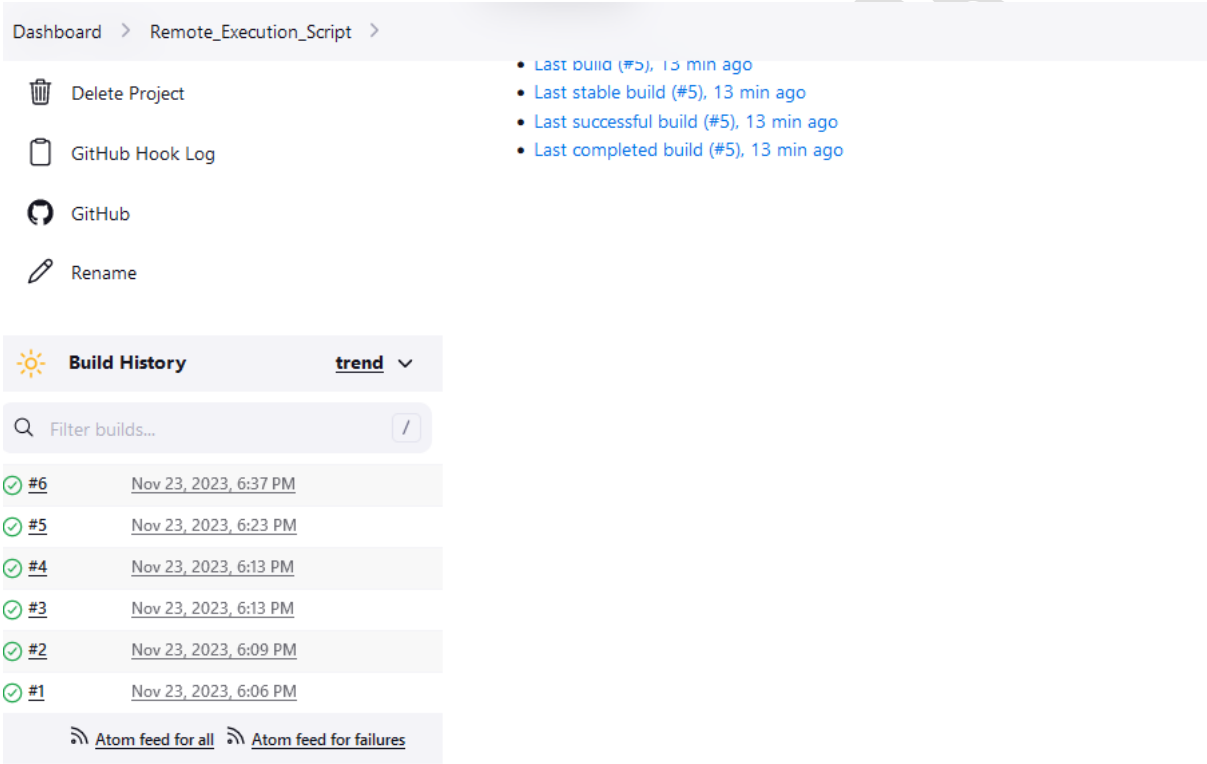
```
chmod 600 /home/ec2-user/.ssh/id_rsa
```

To ensure that the script runs fine you can test it on the ec2 directly by using this command:

2.4 Test Deployment

```
./deploy.sh
```

Check the output and ensure no error is encountered which will mean the job will be executed just fine on Jenkins as seen in the build output below:



3. Conclusion

I hope the steps in this lab helped you get a better understanding of the CI/CD process and setting up the pipeline using industry-standard tools. This automated workflow not only improves development efficiency but also ensures the reliability and consistency of your application deployments.