PG DO – Devops capstone project

1.0

|  |  |
| --- | --- |
| VERSION: | 1.0 |
| DATE | 26 February 2022 |

DOCUMENT HISTORY

|  |  |  |  |
| --- | --- | --- | --- |
| **Version Number** | **Issue Date** | **Change by** | **Reason** |
| 1.0 | 26 February 2022 | Triston Claydon | Initial creation |

INTRODUCTION

A leading US healthcare company, Aetna, with a large IT structure had a 12-week release cycle and their business was impacted due to the legacy process.

To gain true business value through faster feature releases, better service quality, and cost optimization, they wanted to adopt agility in their build and release process.

# PROJECT OBJECTIVES

The objective is to implement iterative deployments, continuous innovation, and automated testing through the assistance of the strategy.

## Implementation requirements:

Set up the Jenkins server in master or slave architecture

* Install and configure the Jenkins architecture on AWS instance
* Use the required plugins to run the build creation on a containerized platform
* Create and run the Docker image which will have the application artifacts
* Execute the automated tests on the created build
* Create your private repository and push the Docker image into the repository
* Expose the application on the respective ports so that the user can access the deployed application
* Remove container stack after completing the job

## The following tools must be used:

* EC2
* Jenkins
* Docker
* Git

# ASSUMPTIONS

|  |  |
| --- | --- |
| No | Description |
| A1 | Linux is pre-installed on the Jenkins Master |
| A2 | Linux is pre-installed on the Jenkins Slave |
| A3 | Linux is pre-installed on the developer workstation |
| A4 | A Web browser is installed on the developer workstation and has access to the internet |
| A5 | The application code has been provided by the developers |
| A6 | An AWS, GitHub and Docker Hub account |

CONCEPTS USED IN THE PROJECT

There are a number of tools that we can use to improve and speed up the legacy 12-week release cycle that Aetna currently follow. These are designed to work together to achieve the goal.

Agile

Agile is a project management methodology that takes an iterative approach to software development and allows for a constant delivery of value to the customer.

The development is broken into Epic’s then into User Stories, from there the work can be prioritized by the product owner and then worked on by development team with each user story being worked on within a time box or sprint.

During the time box\sprint the user story will be developed all the way through to the delivery of a release.

Following this method allows for the continuous delivery of value throughout the project and also makes it flexible to change.

Devops

To make the most of Agile, a new way of delivering infrastructure had to be developed, this way of working is called Devops.

Devops is a compound of development and operations and enables these two formally siloed roles to coordinate to increase productivity, deliver value to the customer faster and with greater customer satisfaction.

Devops has its own life cycle which maps out the delivery of the infrastructure and code release. The steps in the life cycle are:

## Plan

Identify the business requirement and create a project roadmap.

## Code

Code the development, use tools such as Java, .Net and Git.

## Build

Build the code with tools such as Maven and Gradle.

## Test

Test the code using tools such a JUnit or Selenium. This includes acceptance testing, unit testing, integration and security testing.

## Release

Once all the tests have been passed, operational teams begin the move of the build into the production environments.

## Deploy

During this stage Infrastructure as Code is used to build the production environment and then release the build using tools such as Jenkins.

## Operate

The release is Live with the operations teams continuing to support the environment, configuring and provisioning services using tools such as Ansible and Puppet.

## Monitor

During the monitor phase the pipelines, applications and the infrastructure is all monitored with tools such as New Relic. This is end to end monitoring and helps find issues and bottlenecks quickly which is fed back to the devops\development team.

# Infrastructure as Code

The traditional method of managing infrastructure also needed to be revisited to allow for an Agile approach and out of this Infrastructure as Code was born.

Infrastructure as code is the management of infrastructure: servers, networks, load balancers, firewalls, switches etc: through code. It uses API’s and technologies like Terraform, Jenkins, Puppet, Ansible and Chef to achieve this.

By using code, we can deliver infrastructure that consistently meets our designs and allows for repeatability whilst reducing the time it takes to deliver infrastructure thereby reducing costs and increasing customer satisfaction.

# SCM

Now that our infrastructure is being delivered via code, we need to store that code in a source control management system repository.

A SCM will allow the development team to work on the code in a collaborative fashion. The development team will also adopt a branching and merging strategies such as Git Flow or Trunk-based development.

CI\CD Pipelines

Pipelines are a way to automate the delivery of the infrastructure as code. We will adopt the use of Continuous integration and Continuous Delivery\Deployment pipelines.

Containers

Containers are a set of platform as service that use OS-Level virtualization to deliver software in packages called containers.

This greatly increases the speed at which software can be delivered and allows for delivery strategies such as blue green deployment.

# The Aetna Project Overview

For this project I will:

Create a Continuous Integration and Delivery pipeline that will:

* Detect if the code has been released
* Build the code
* Unit test the code
* Package the code into a container
* Function test the container
* Deliver the code on success to private container repository

## Technology:

* Use Jenkins as our pipelining tool
* Use Docker as our container platform
* Use Public cloud to host the Jenkins Master, Slave and Docker platform
* Git and GitHub as the code repository
* Maven to build and package the code
* Java to run the code
* Docker Hub for the private image store

## Project Steps:

* Create a Git repository
* Create a private Docker repository
* Build Jenkins Master and Slave on AWS
* Install Docker onto the Jenkins Slave
* Create Jenkins build pipeline
* Test the pipeline
* Conclusion

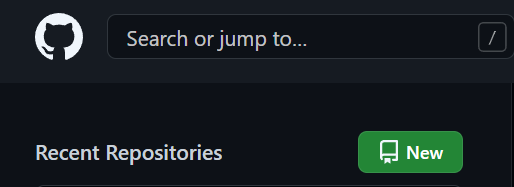
# Create Git Repositories

We will create two repositories: A local git repository on the developer workstation where code can be pulled and pushed to a remote shared Git Hub repository.

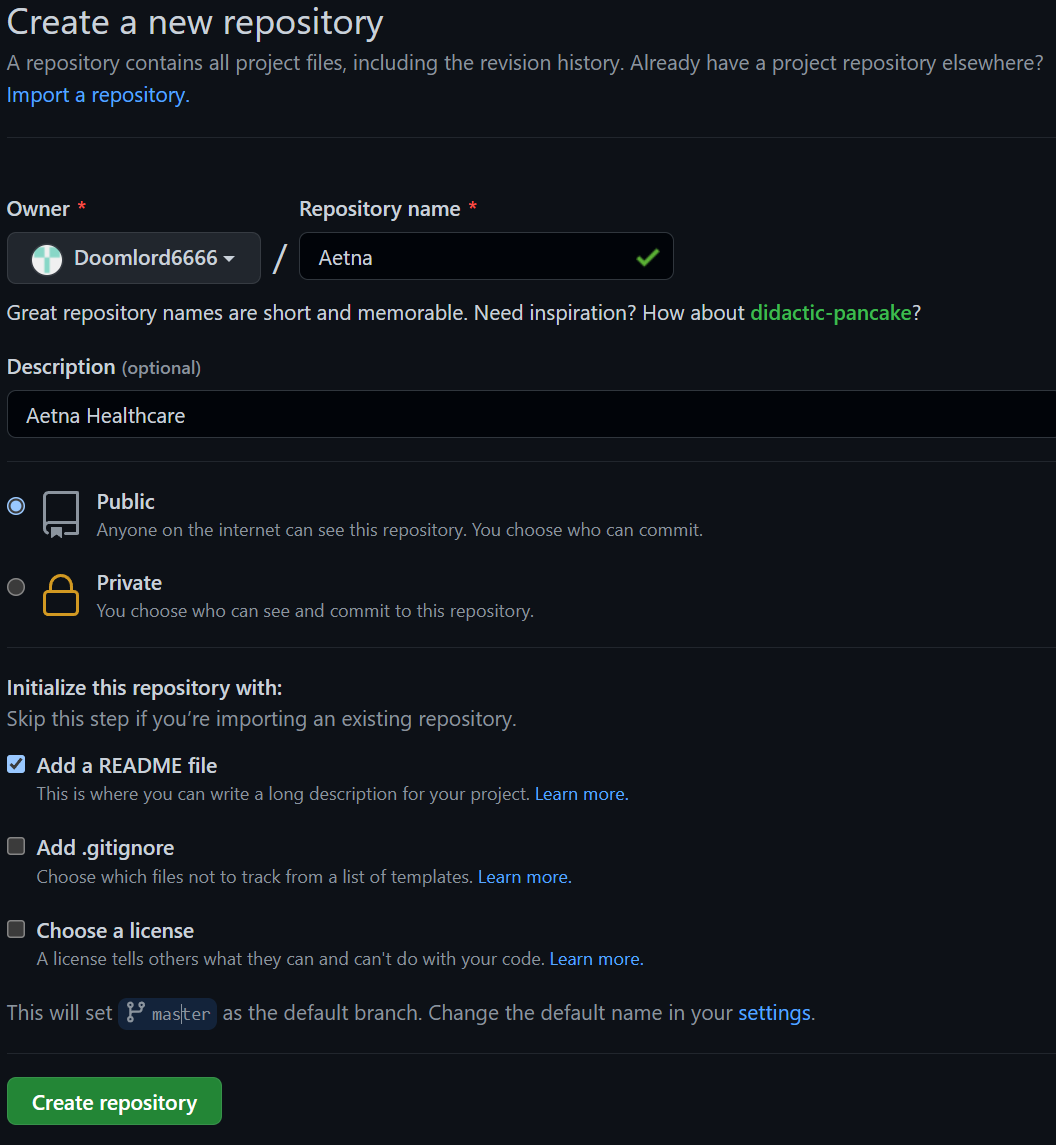
This will allow multiple developers to work on the code and present it for use by the CI\CD tooling.

## Creating the remote repository

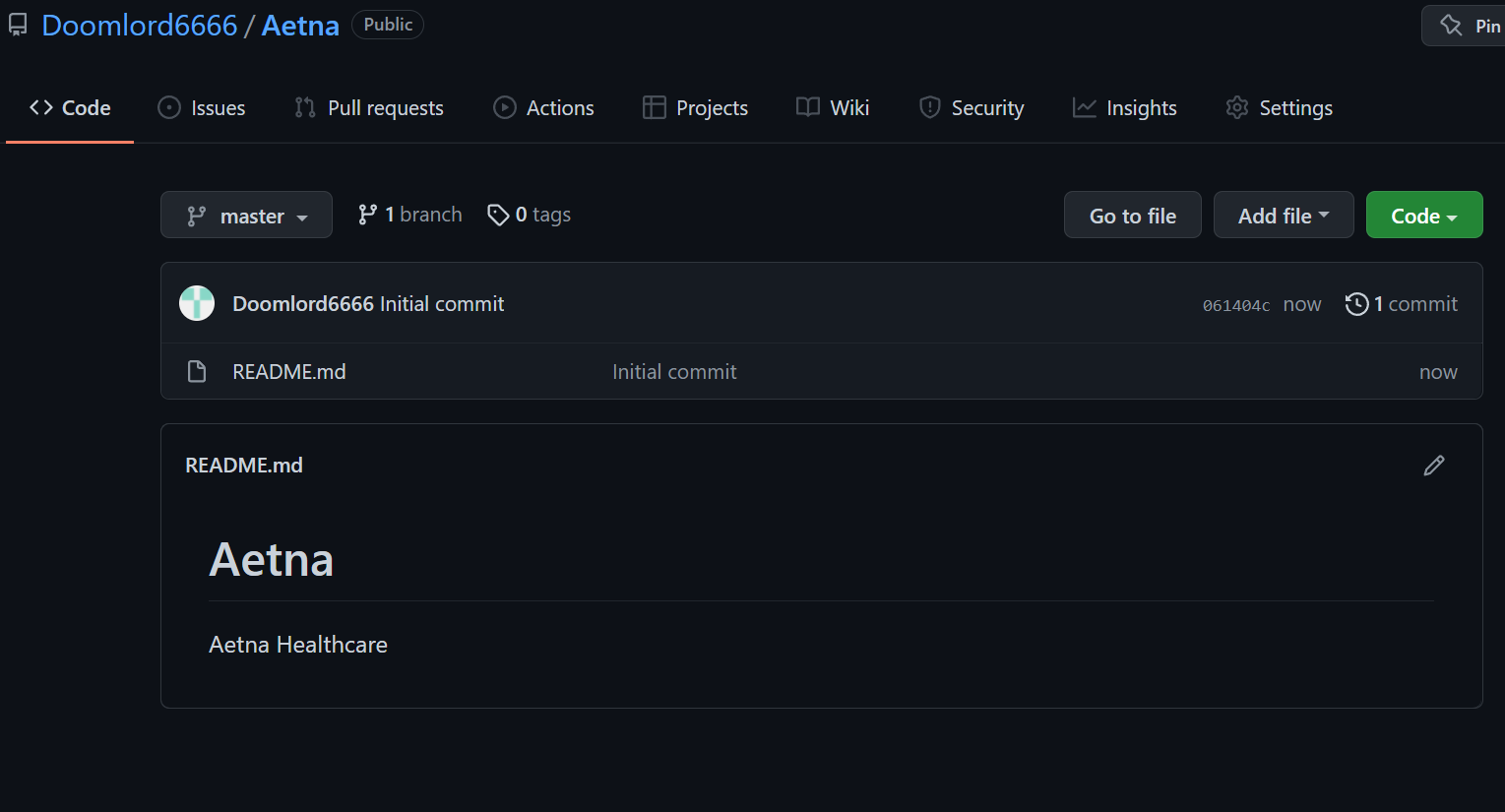
Froom the developer workstation log into <https://github.com/>



Select “New”



Enter the following details for Aetna health care and press the “create repository” button.



A new empty repository has been created and is ready for use on the following address:

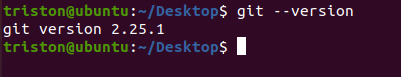
<https://github.com/Doomlord6666/Aetna.git>

git@github.com:Doomlord6666/Aetna.git

## Creating the local repository

On the developer workstation run the following commands to install Git:

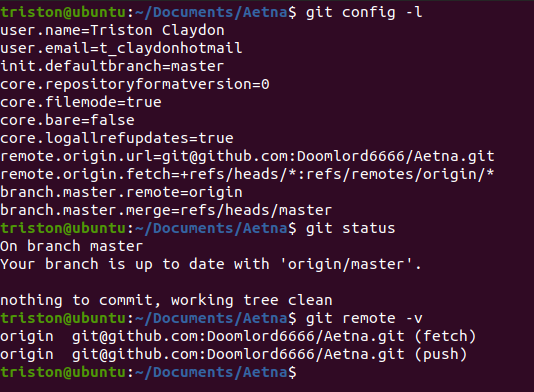
* Sudo apt-get update
* Sudo apt-get install git
* Git –version



Here we can see that git has been installed and is ready for use.

On the developer workstation run the following commands to create our new local git repository for Aetna healthcare:

* Generate ssh key and add to git hub following these instructions: [Generating a new SSH key](https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent)
* cd /home/triston/Documents
* git config --global user.name “Triston Claydon”
* git config --global user.email “t\_claydon@hotmail.com”
* git config --global init.defaultbranch master
* git clone git@github.com:Doomlord6666/Aetna.git
* cd Aetna
* git config -l
* git status
* git remote -v

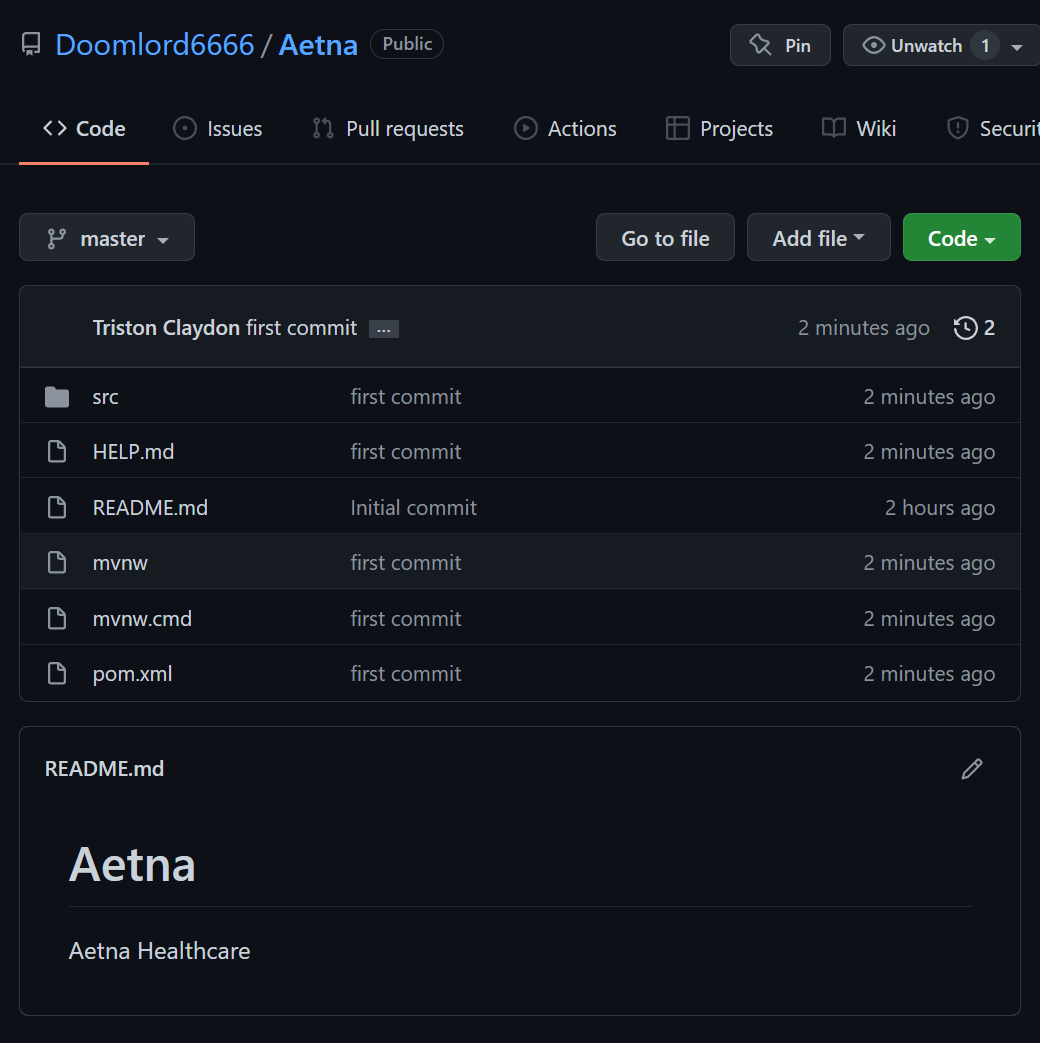


Here we can see the local repository has been created, is up to date and connected to the remote git repository.

## Add code the repository and push to the remote repository

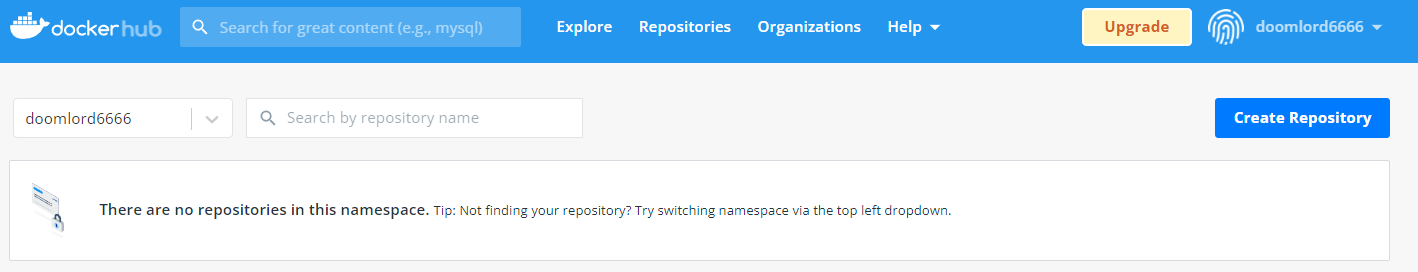
For the initial commit we need to:

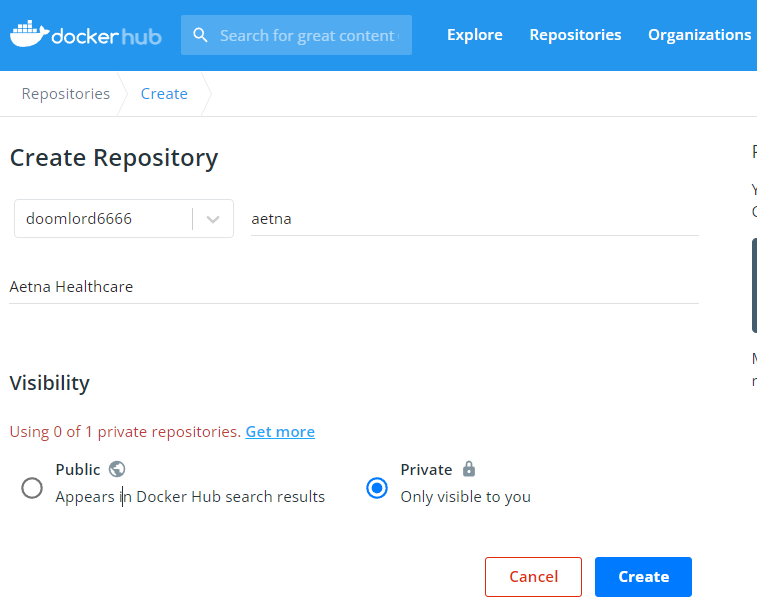
* Copy the application code into the local git repository
* git add \*
* git commit –m “initial commit”
* git pull
* git push

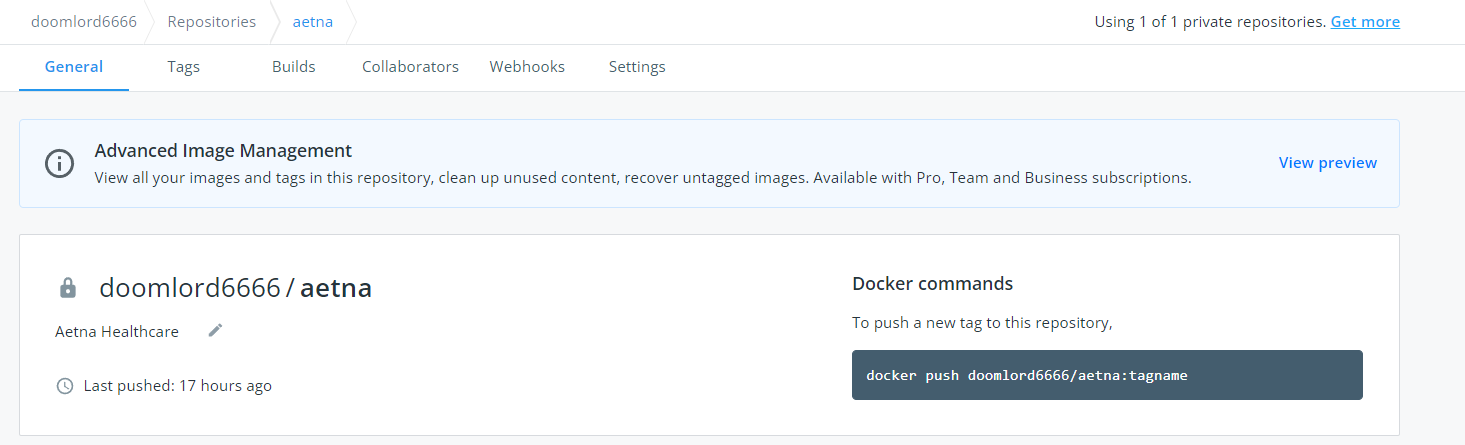


Here we can see the first commit of the code from our local git repository to our remote git repository

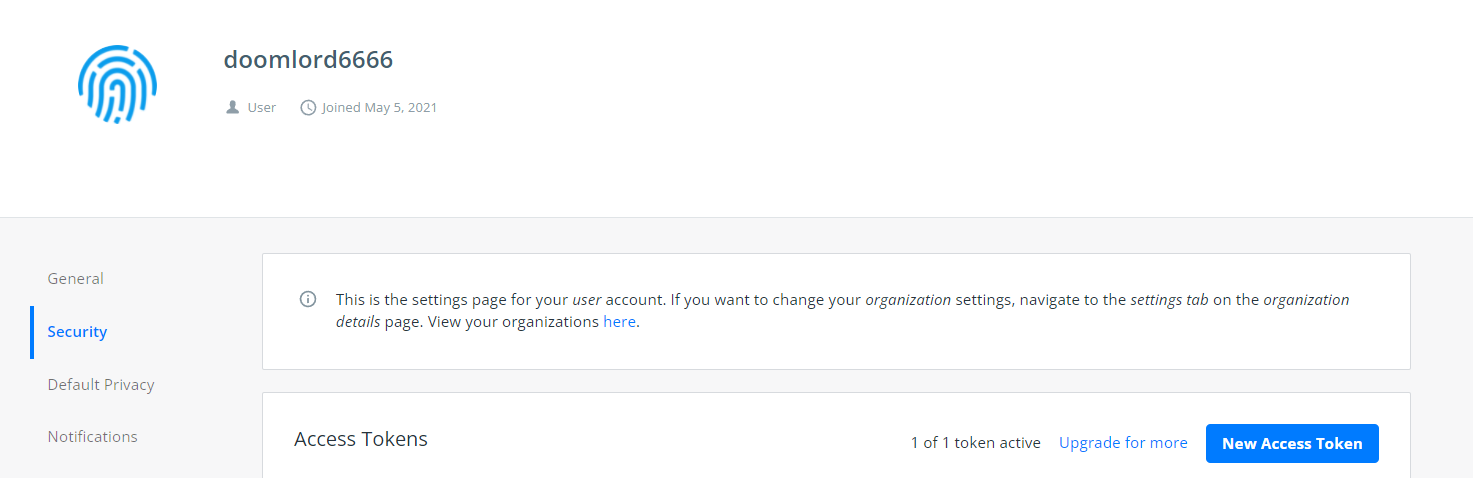
# Create a private Docker repository

  
Log into dockerhub and press the “Create Repository” button.

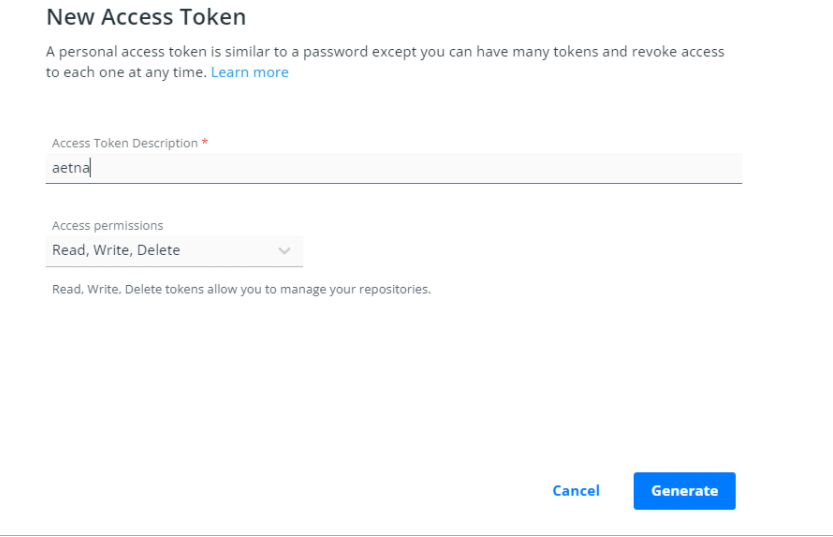
  
Name the repository “aetna”, select the “Private radio dial” and press the “Create” button.



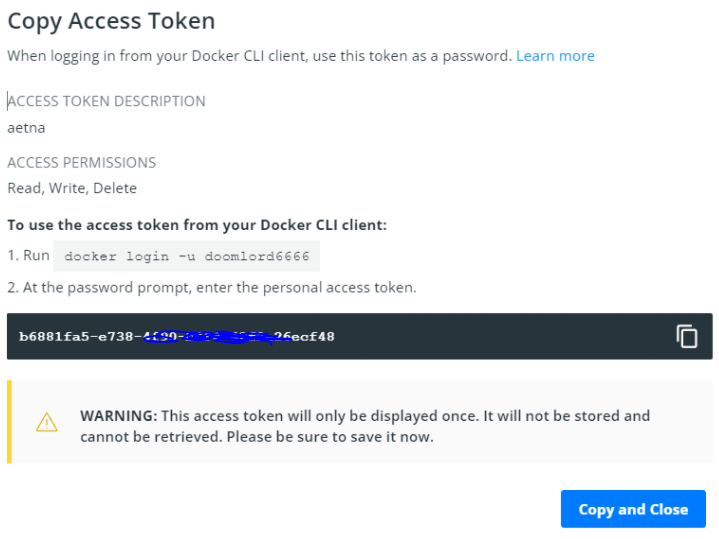
The private repository has been created.



Now to create an access token for our pipeline to use later: go to the user settings and press the “New Access Token” button.



Enter the token details and press the “Generate” button.



The token has been created, record it for later use.

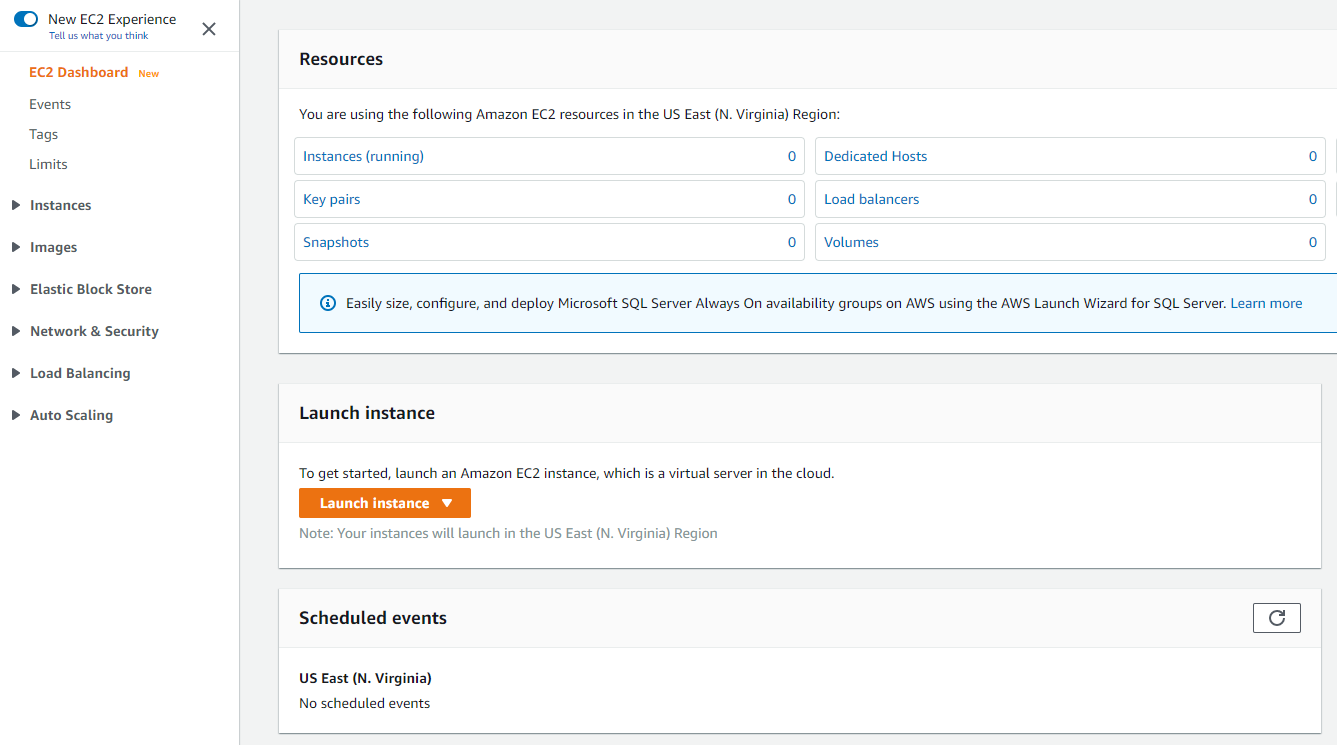
# Build Jenkins Master and Slave

We will use Amazon AWS to host both the Jenkins Master and the Jenkins Slave and will undertake the following steps to achieve this:

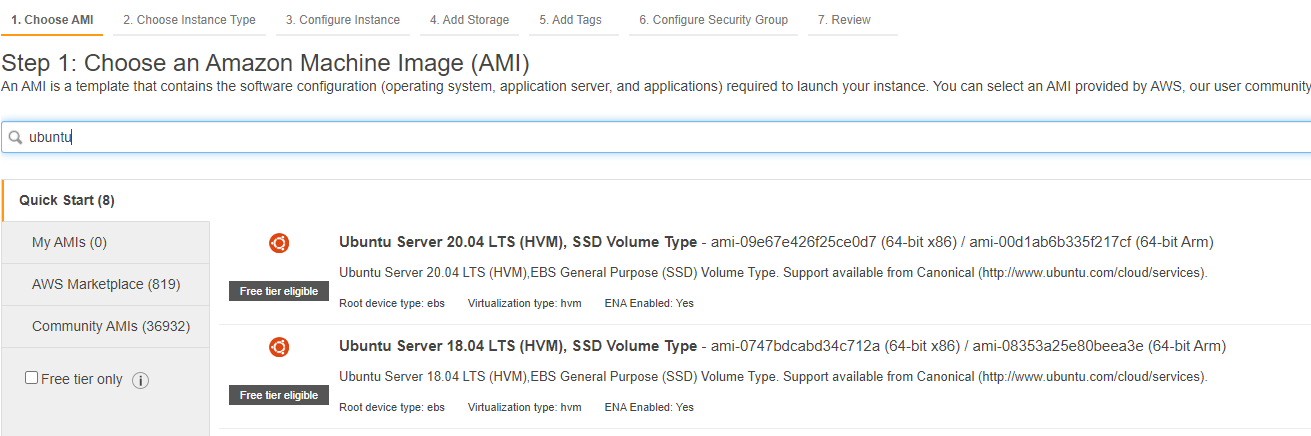
* Provision 2 AWS EC2 Instances.
* Install Jenkins on the two machines.
* Configure machine one to be the Jenkins master
* Configure Jenkins on machine two to be the Jenkins Slave

## Provision 2 AWS EC2 Instances

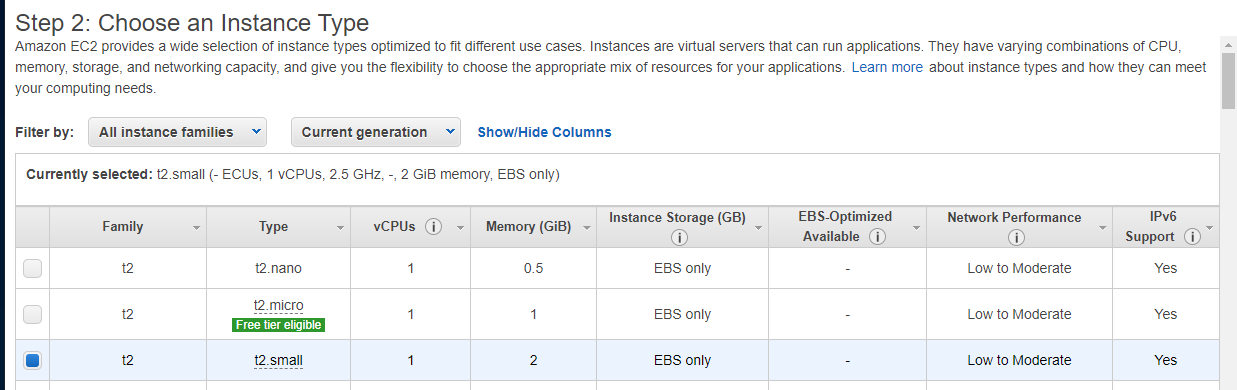
Log into AWS Dashboard: <https://console.aws.amazon.com/ec2/v2/home?region=us-east-1#Home>:”



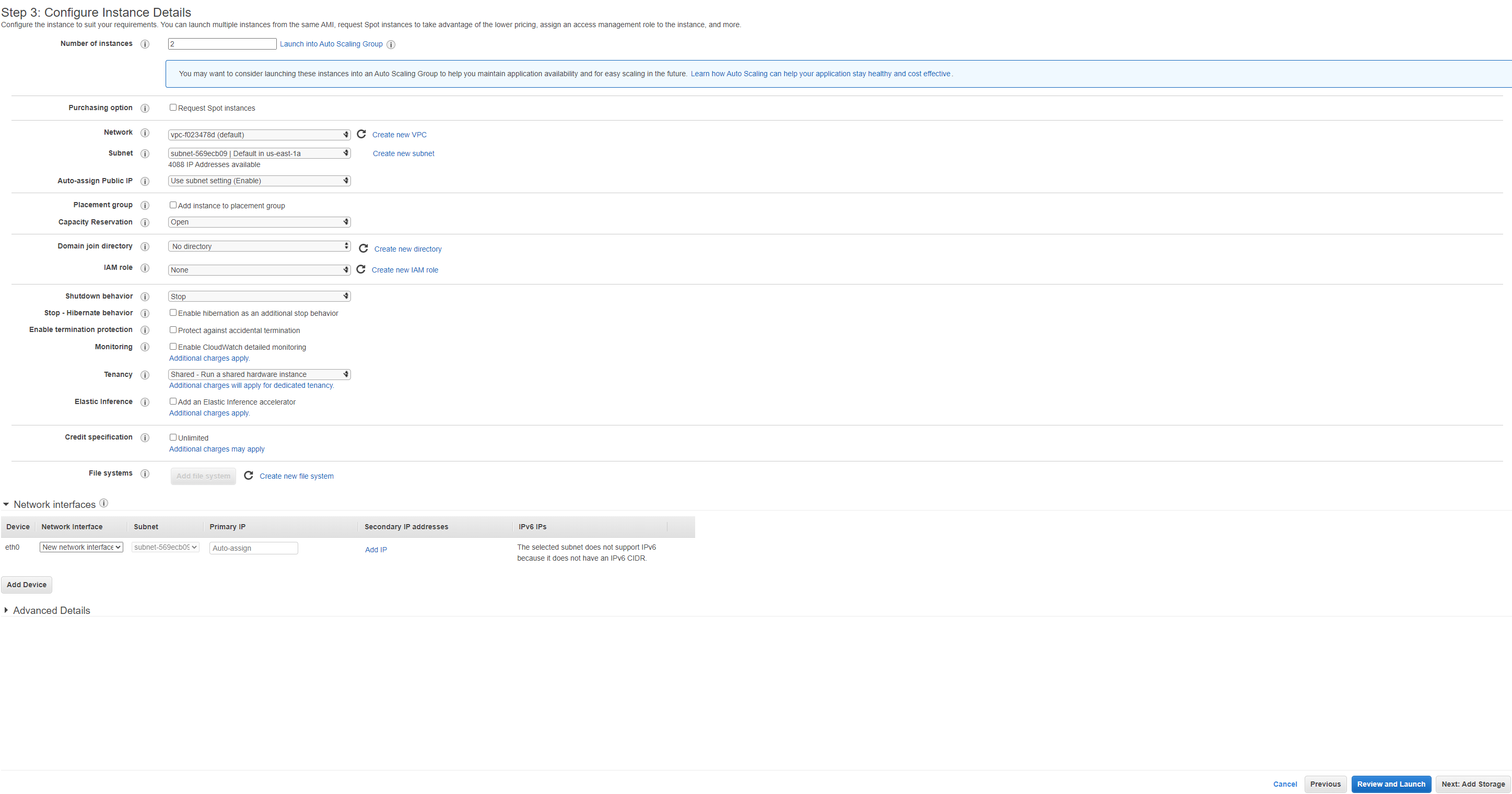
Press the “Launch Instance” button.



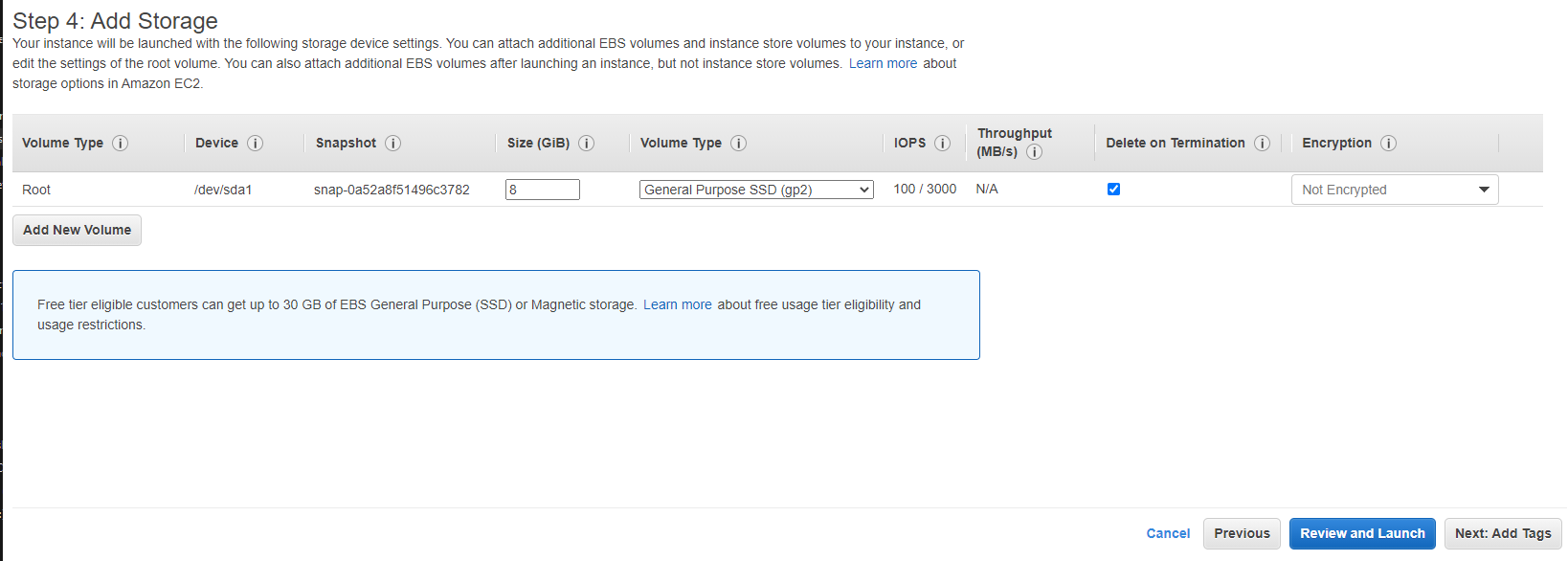
Search for and select an “Ubuntu Server 20.04 LTS (HVM)” server.



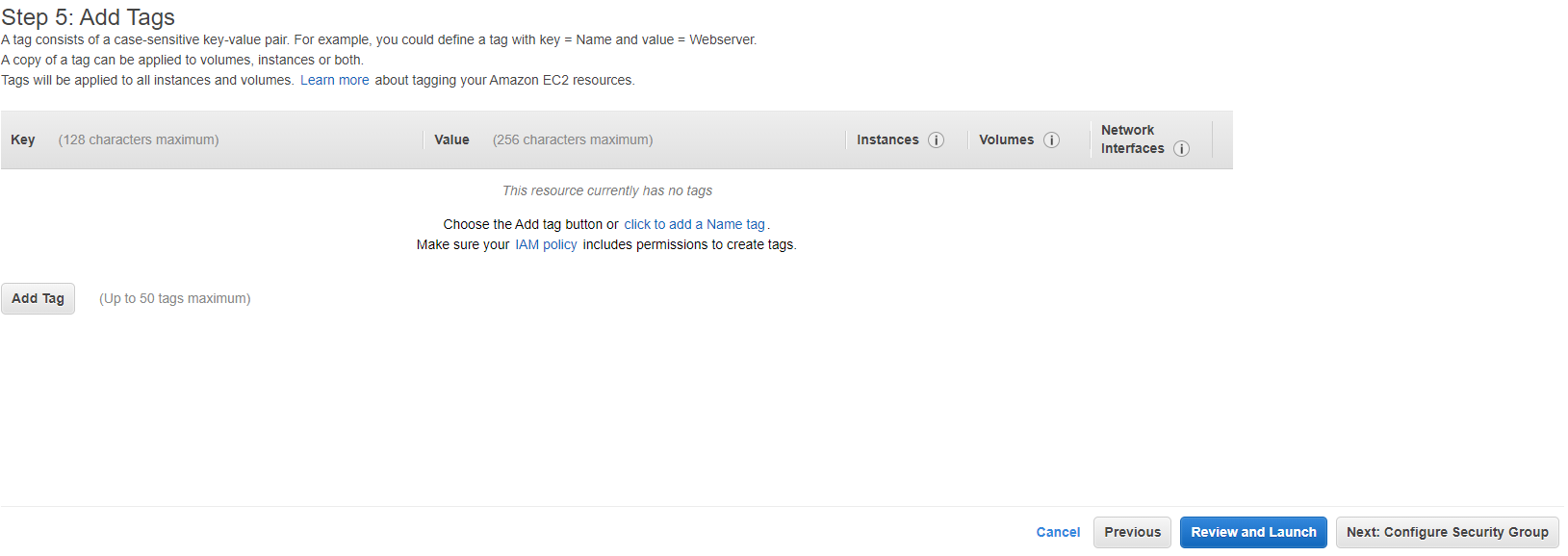
Select the “t2 small” as this meets the requirements and press the “Next: Configure instance details” button.



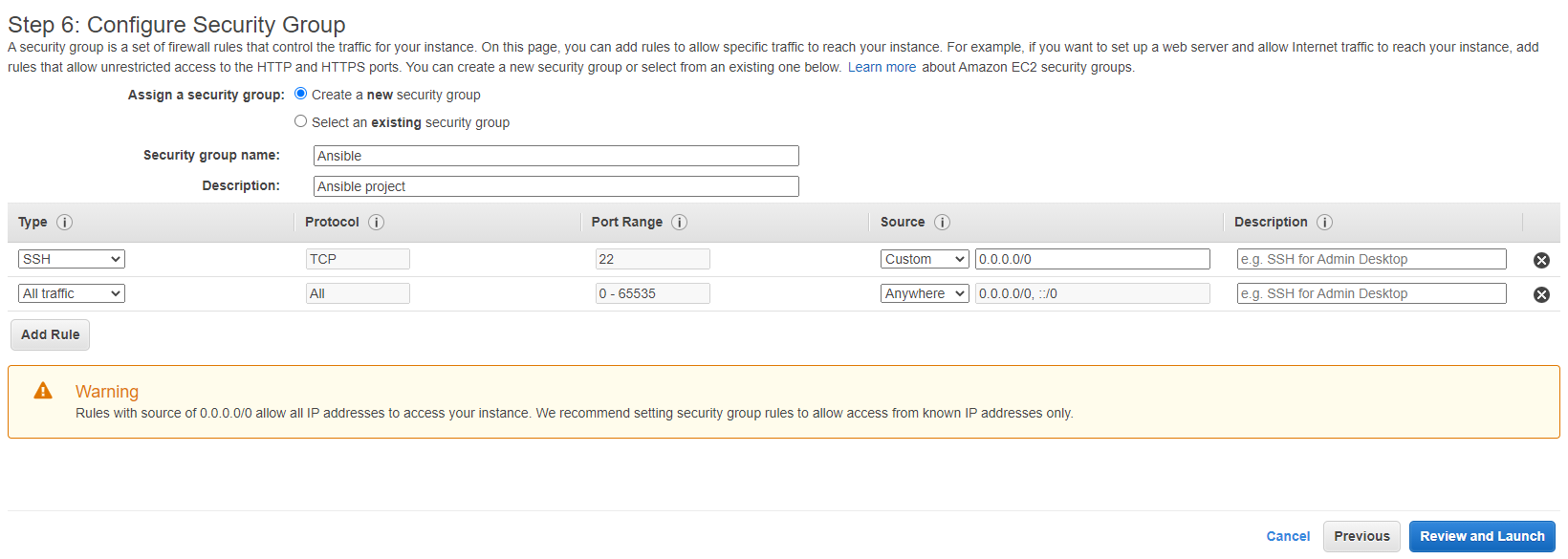
Enter configuration details above and press the “Next: Add Storage” button.



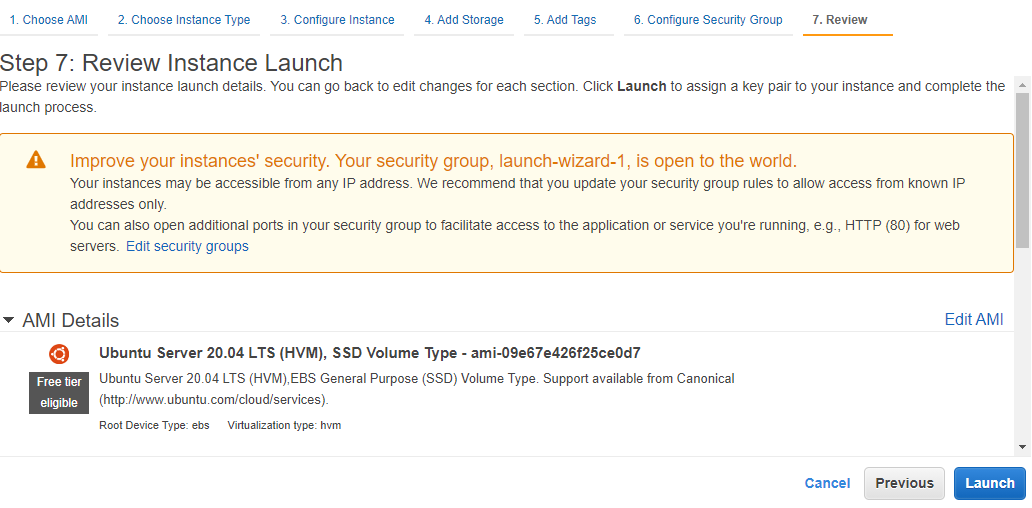
Press the “Next: Add Tags” button.



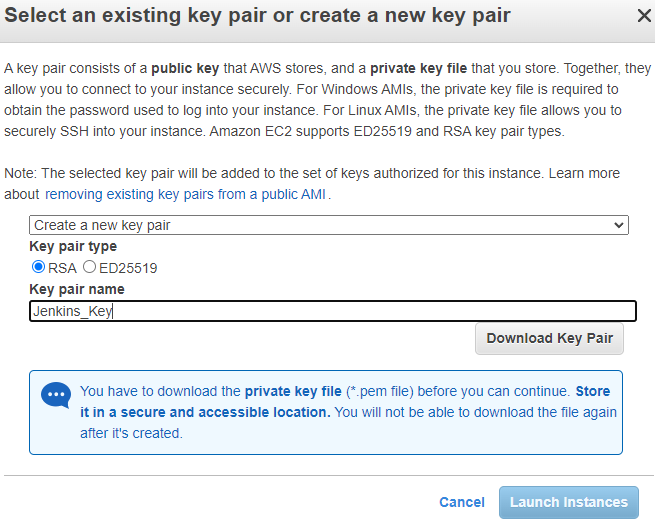
Press the “Next: Configure Security Group” button.



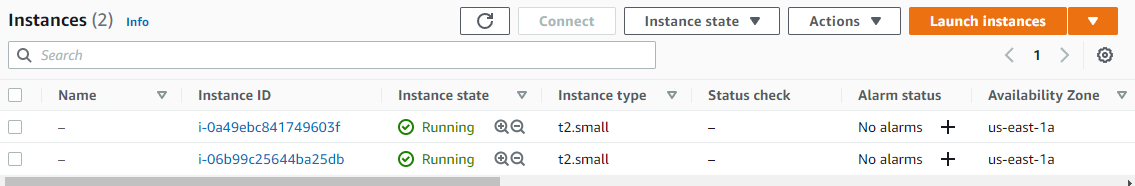
Enter configuration details above. Please note this is OK for our test environment but we would not use these settings in a production environment due to security reasons. Press the “Review and Launch” button.



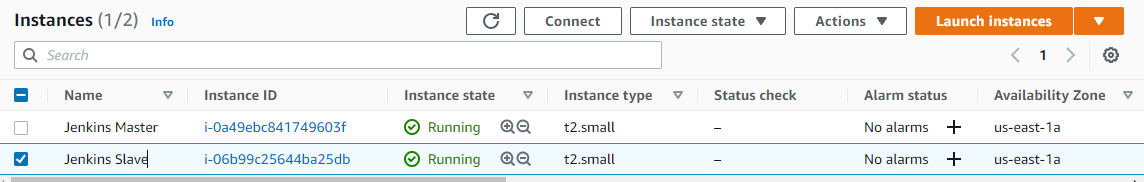
Press the “Launch” button.



Create a new key pair, press the “Download Key Pair” button and then the “Launch instances” button.



Back on the AWS dashboard we can see two instances have been created and are running.



Edit the “Name” to make them more identifiable.

From this point machine one will be referenced as the Jenkins Master and machine two will be referenced as the Jenkins Slave.

# INSTALL AND CONFIGURE JENKINS

As part of the project, we are required to deliver one Jenkins master server and two Jenkins slave servers. To achieve this, we will follow these steps:

* Install Git on Jenkins Master
* Install Java on Jenkins Master and Slaves
* Install Maven on Jenkins Master and Slaves
* Install Jenkins on Jenkins Master
* Configure Jenkins Master
* Configure Jenkins Slave

## Install Git on Jenkins Master



Update the package lists.



Install git.



Verify git has been installed.

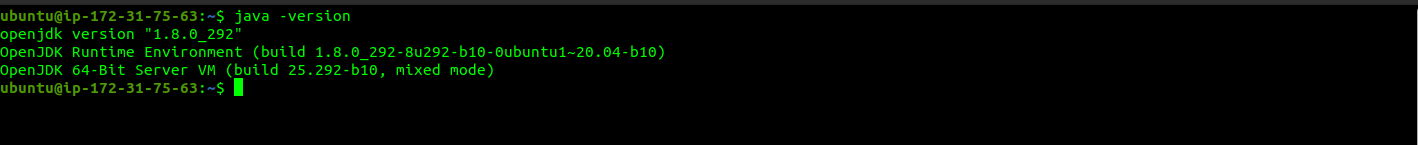
## Install Java on Jenkins Master and Jenkins Slave



Update the package lists.



Install Open Java JDK 8.



Verify Java has been installed.

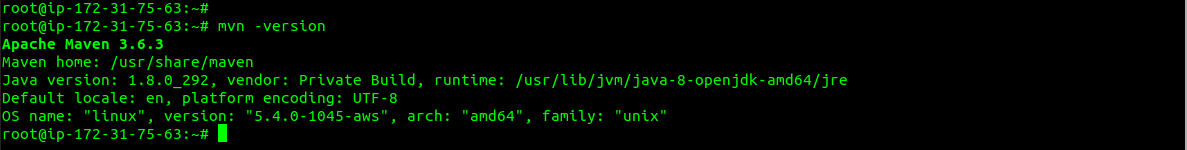
## Install Maven on Jenkins Master and Jenkins Slave



Update the package list.



Install Maven.



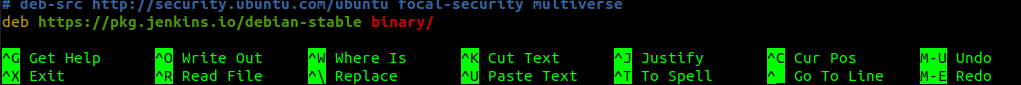
Verify Maven has been installed.

## Install Jenkins on Jenkins Master





add the following entry in your /etc/apt/sources.list.







Update the package lists.



Install Jenkins.

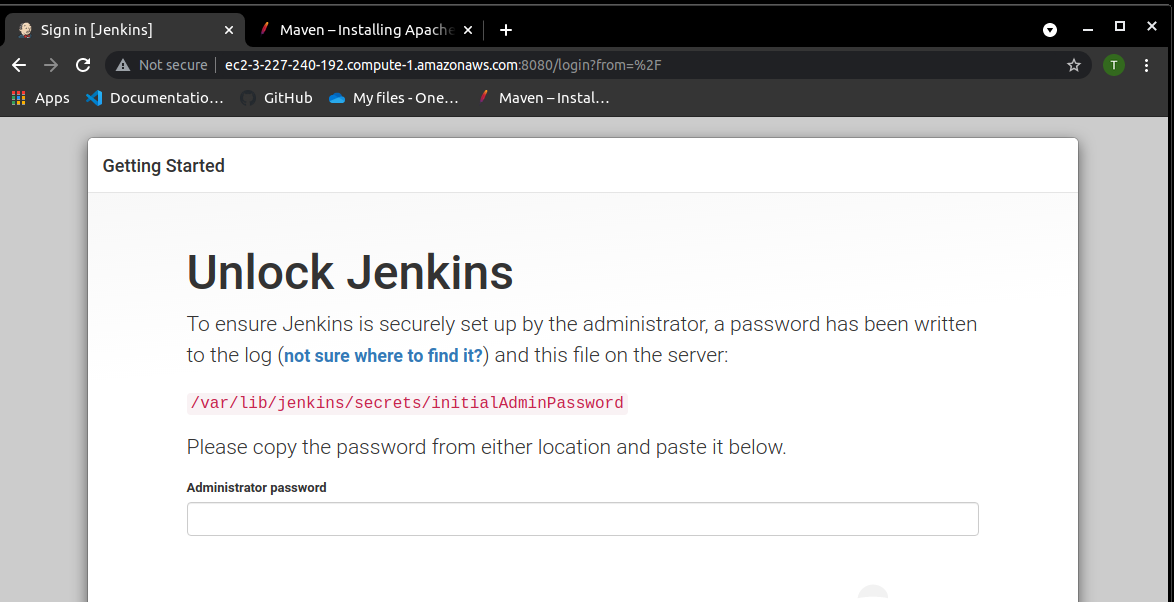


Start the Jenkins service.

Jenkins is now ready to be configured.

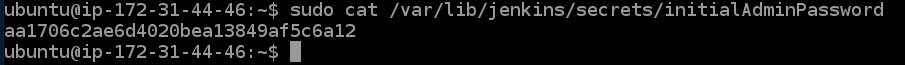
## Configure Jenkins

On the developer machine browse to:

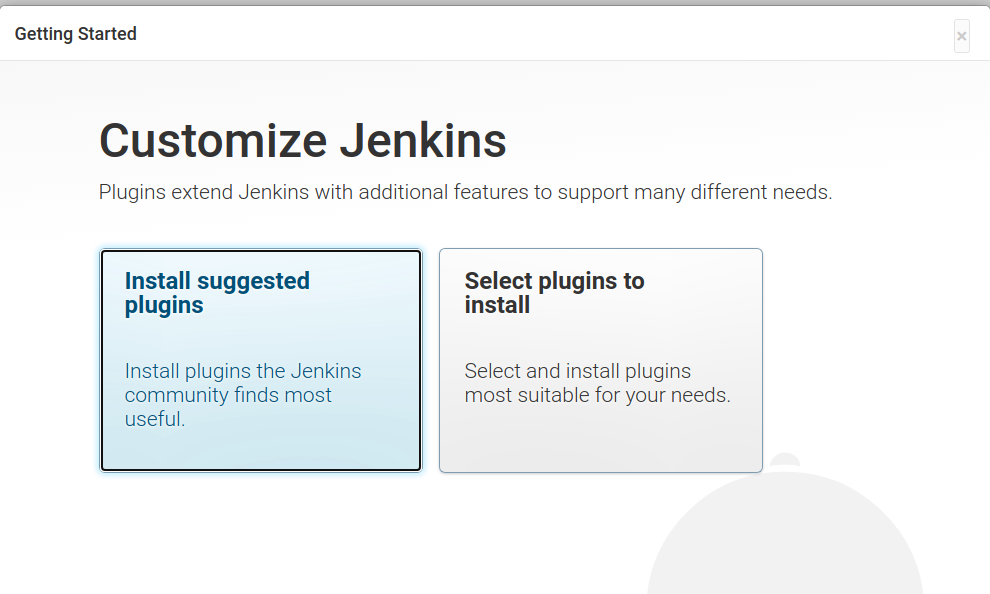


http://ec2-35-173-196-84.compute-1.amazonaws.com:8080/

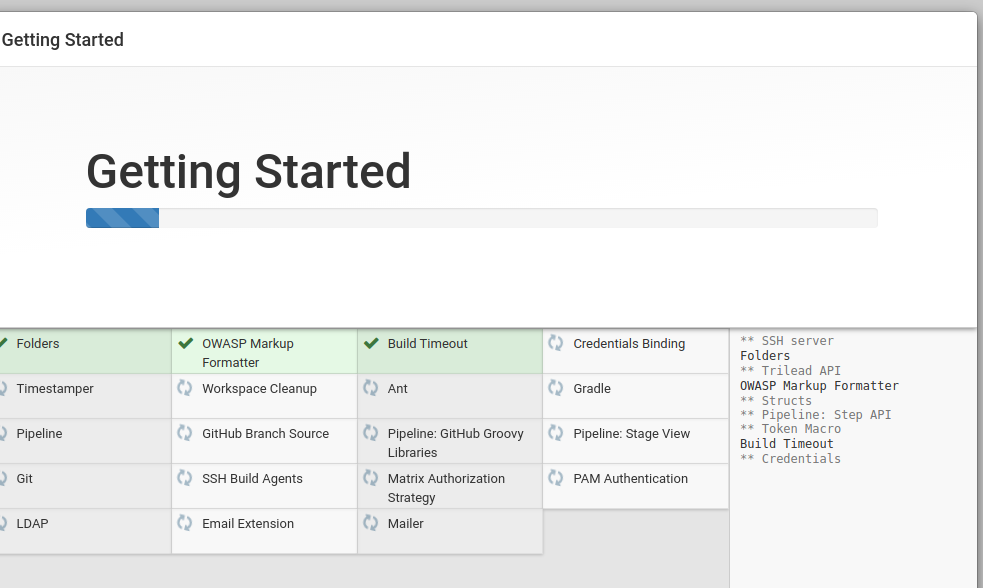
On the Jenkins master:



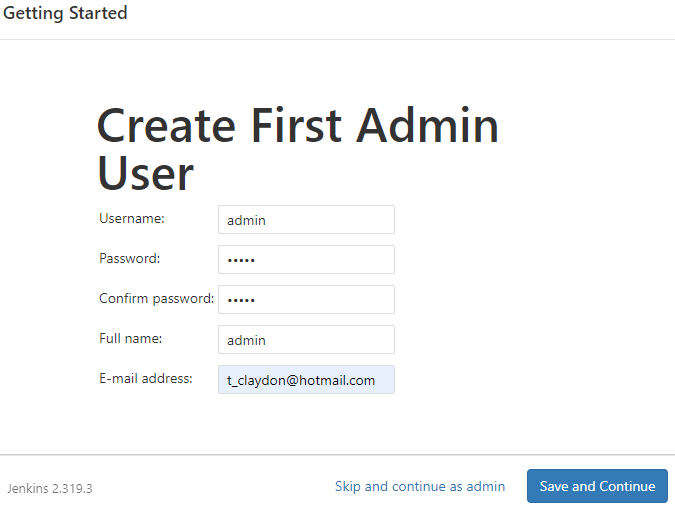
cat the initial password location, copy the password and paste it into the browser on the developer machine.



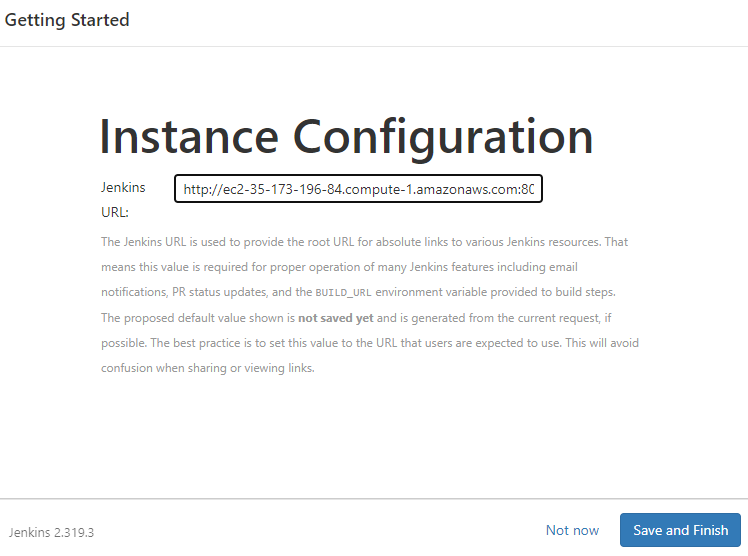
Select “Install suggested plugins”.



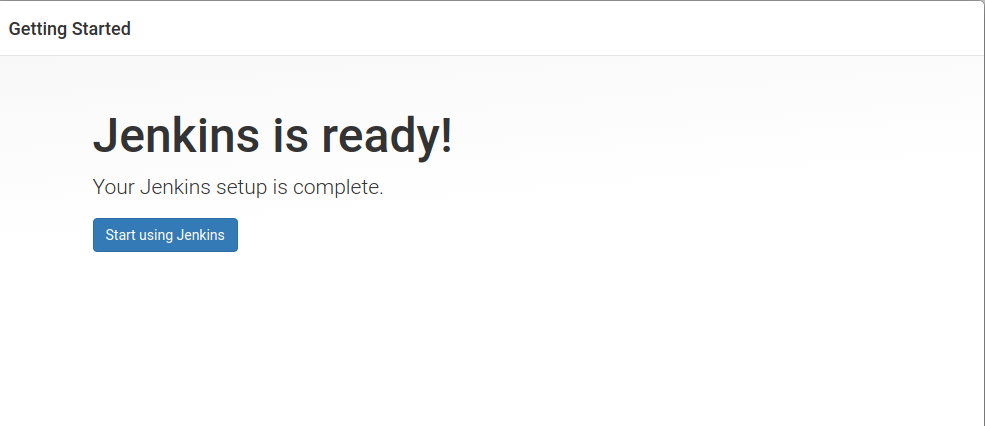
Jenkins will now install a set of default plugins.



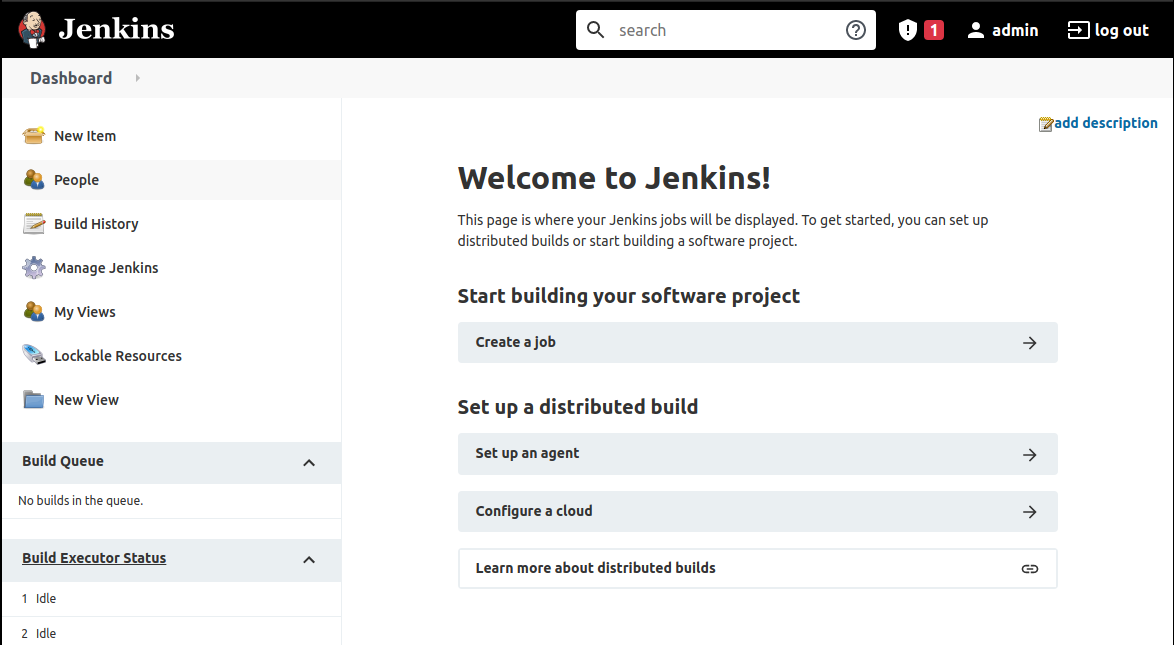
Create first admin user credentials. Store securely for later use. Press the “Save and Continue” button.



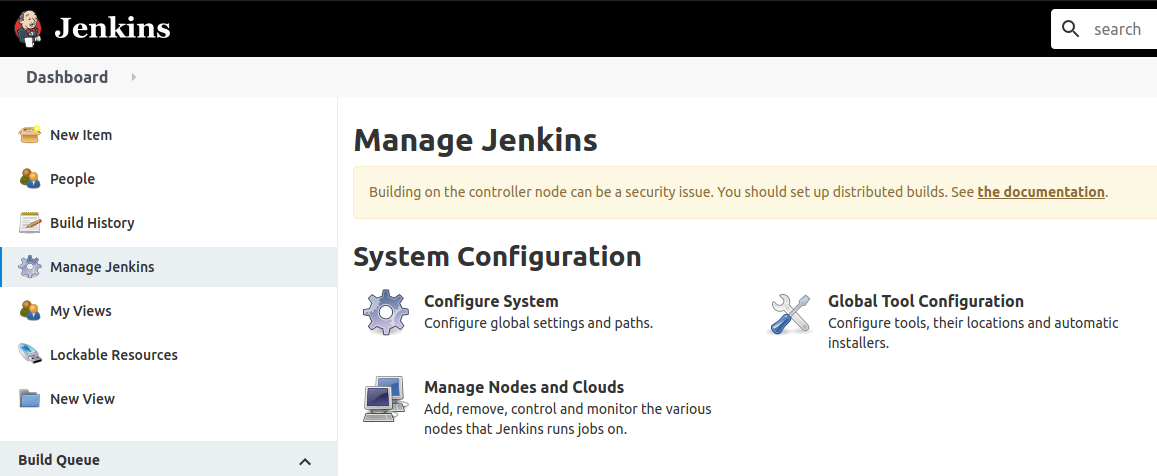
Press “Save and Finish”.



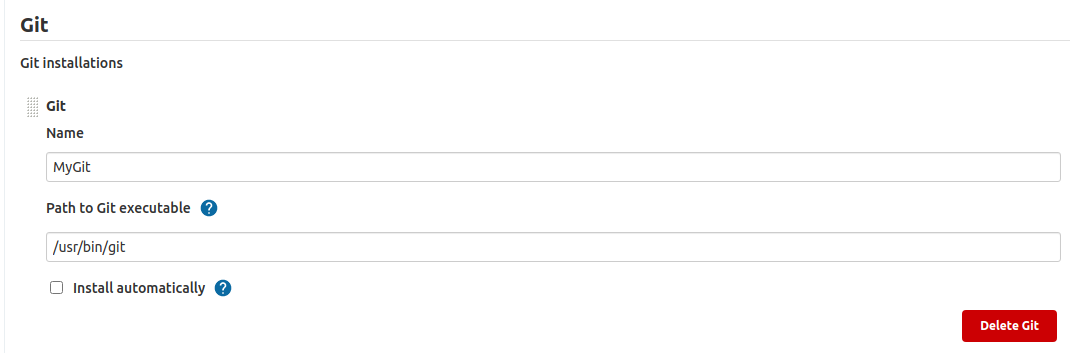
Press “Start using Jenkins”.



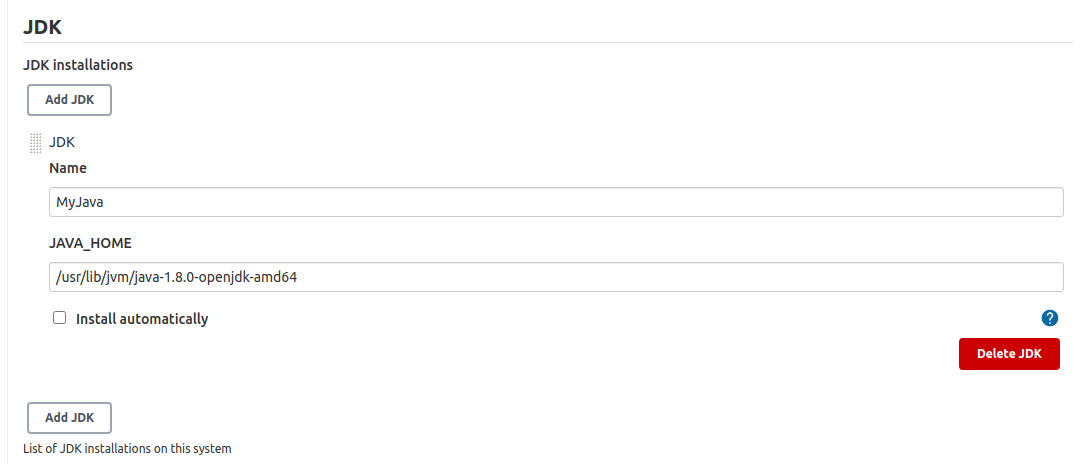
Press “Manage Jenkins”.



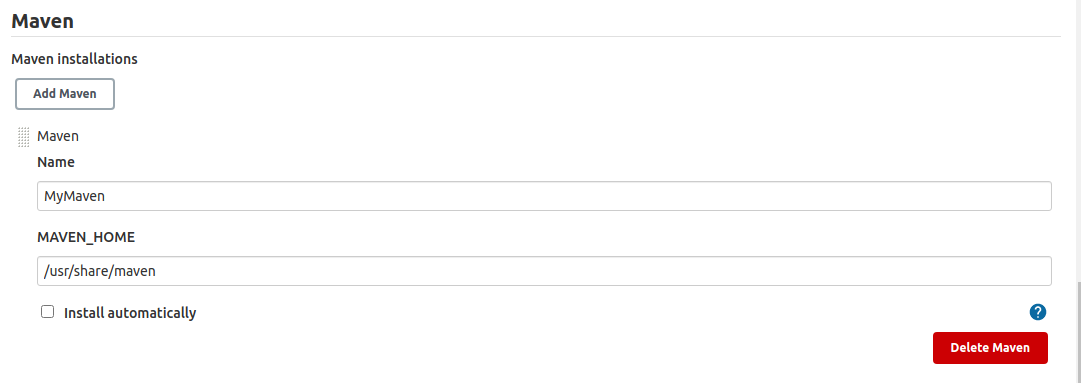
Press “Global Tool Configuration” and configure the following tools:



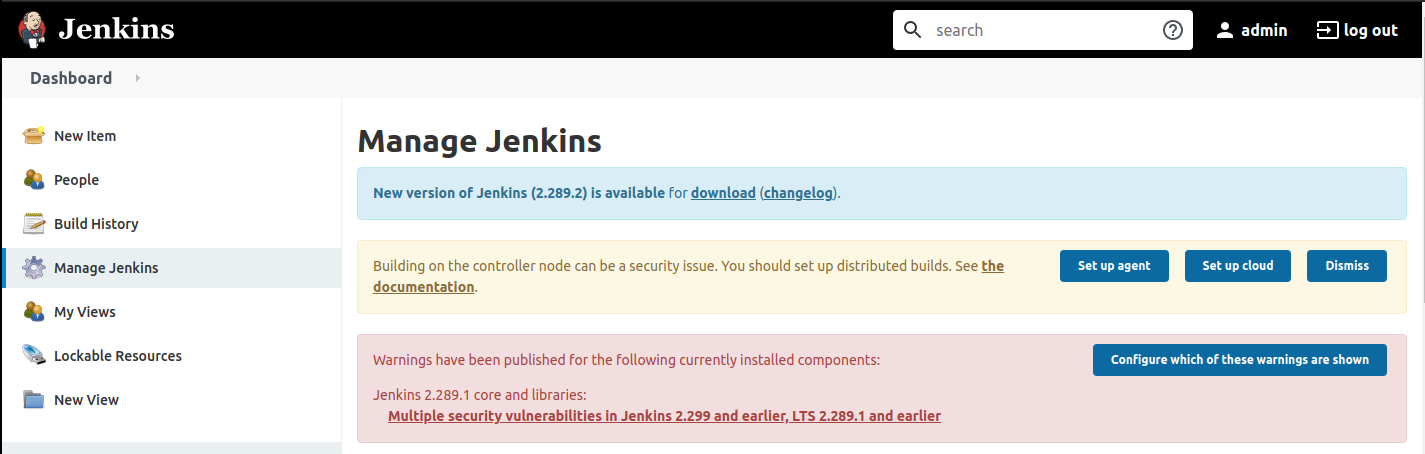
Configure Git installation.



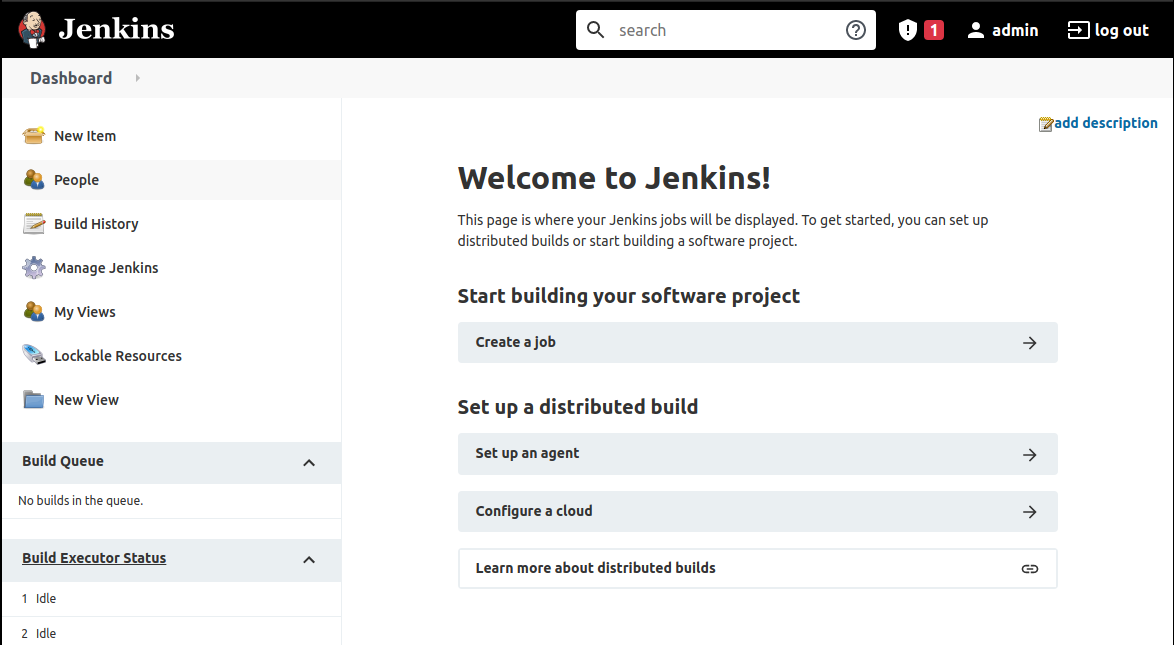
Configure Java installation.



Configure Maven installation and then press the save button.



Press the “Dashboard”.

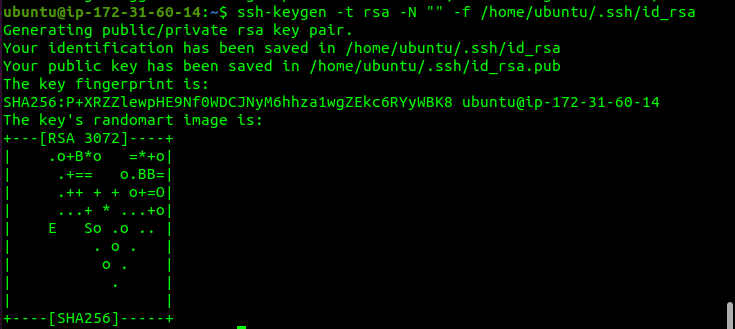


## Configure Jenkins Slave

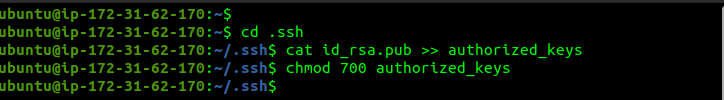
On instruction from the Jenkins Master the Jenkins Slaves will be used to compile, build and test the software. We are required 2 slave servers and the following steps are required to be followed on each server to configure them:

* Create SSH Key on Jenks Slaves
* Add Jenkins Slave SSH keys to Jenkins Master
* Configure Jenkins Slave

### Create SSH Key



Create an SSH Key.



Add SSH Key to authorized keys and set permissions.



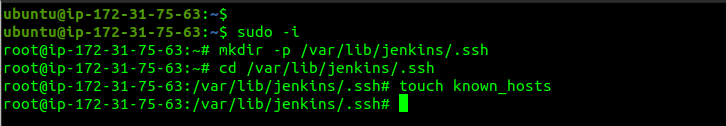
cat id\_rsa and copy the private key into a text editor for later use for configuring the Jenkins Master.



Retrieve server private IP Address for later use configuring the Jenkins Master.

Add Jenkins Slave SSH key to Jenkins Master

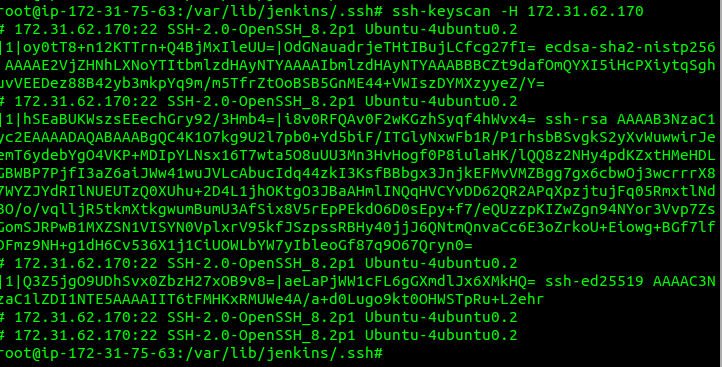
On Master Node:



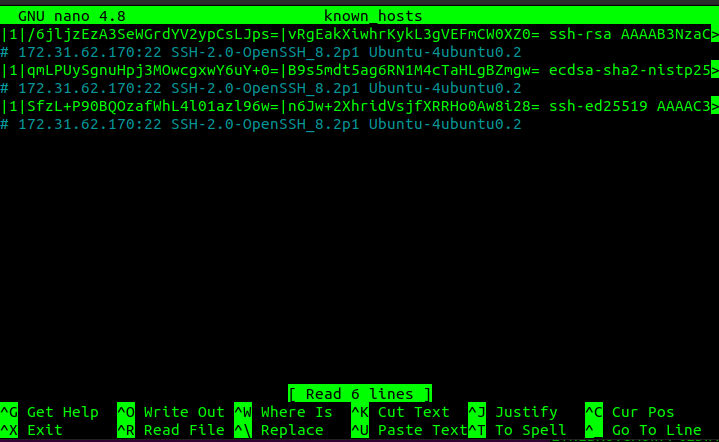


Create an SSH known\_hosts file.

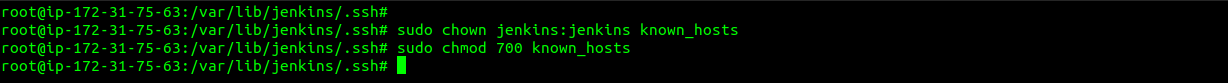
For Jenkins Slave:



Run ssh-key scan using the Jenkins Slave IP and copy the output to the clipboard.

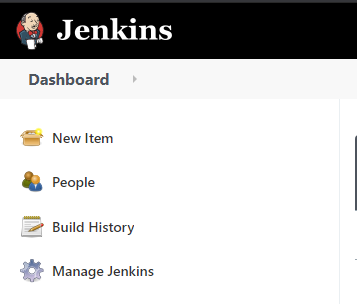


Paste the clipboard into the known\_hosts file and save.

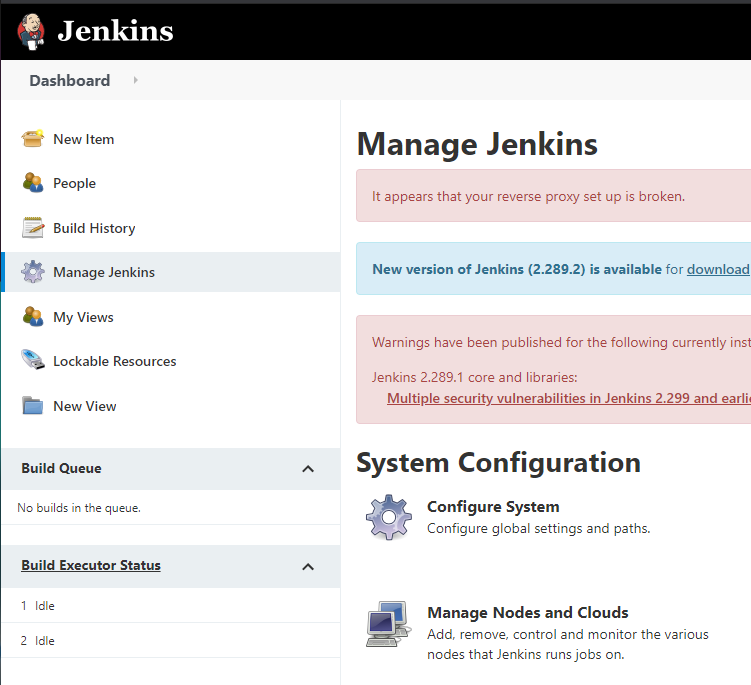


Set ownership and permissions on the known\_hosts file.

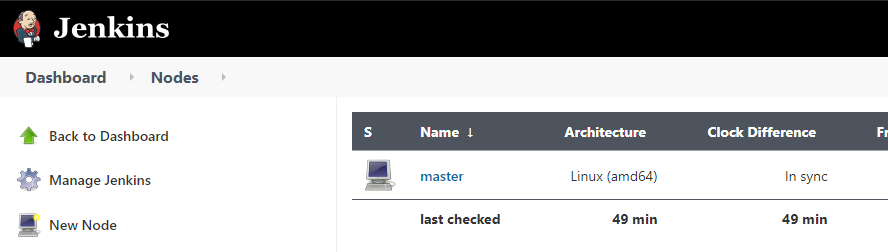
### Configure Jenkins Slaves



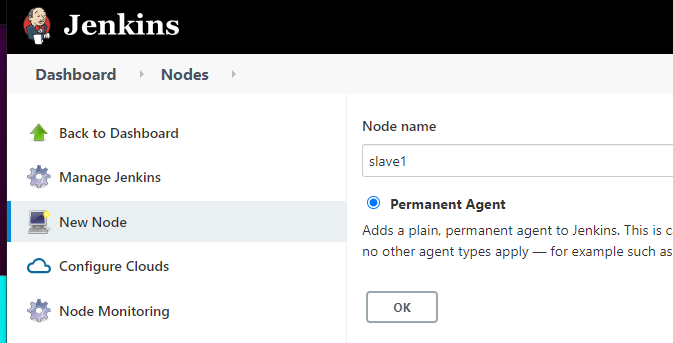
Browse to the Jenkins console and press “Manage Jenkins”.



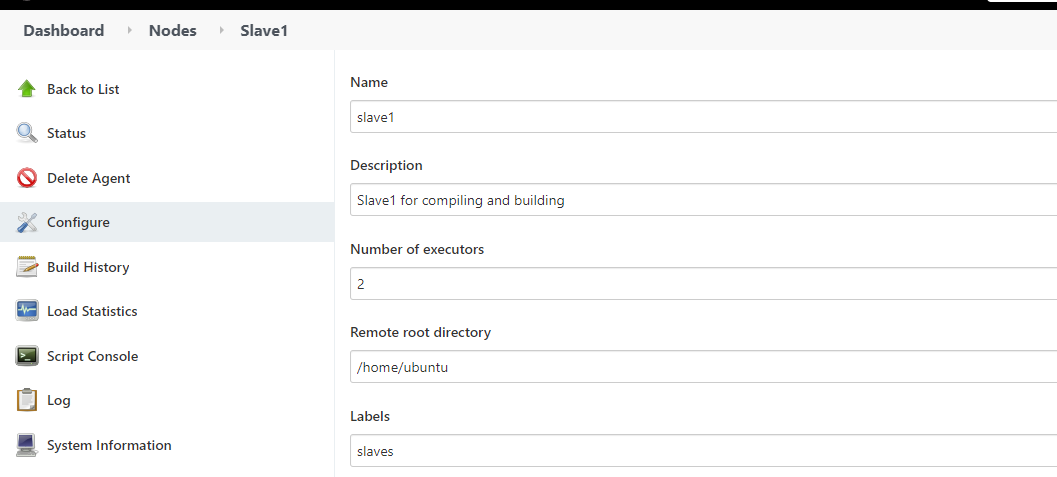
Press “Manage Nodes and Clouds”.



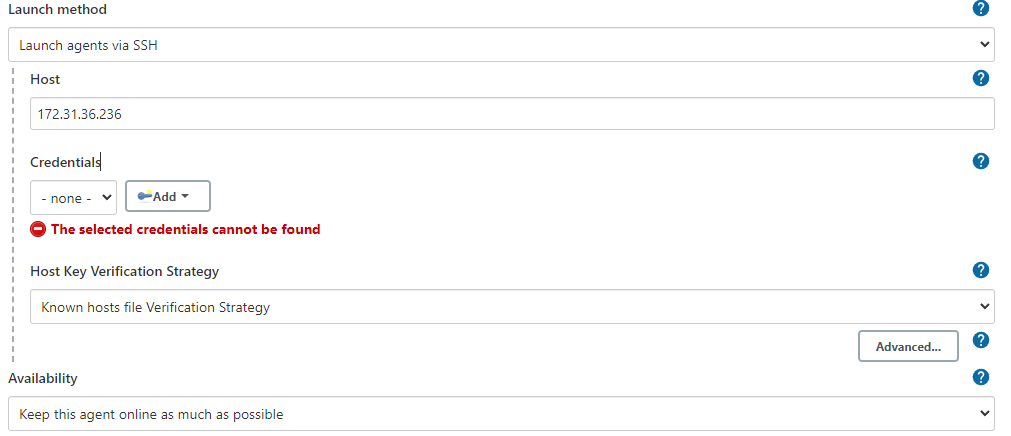
Press “New Node”.



Give the node a name and select the “Permanent Agent” radio button and press “OK”



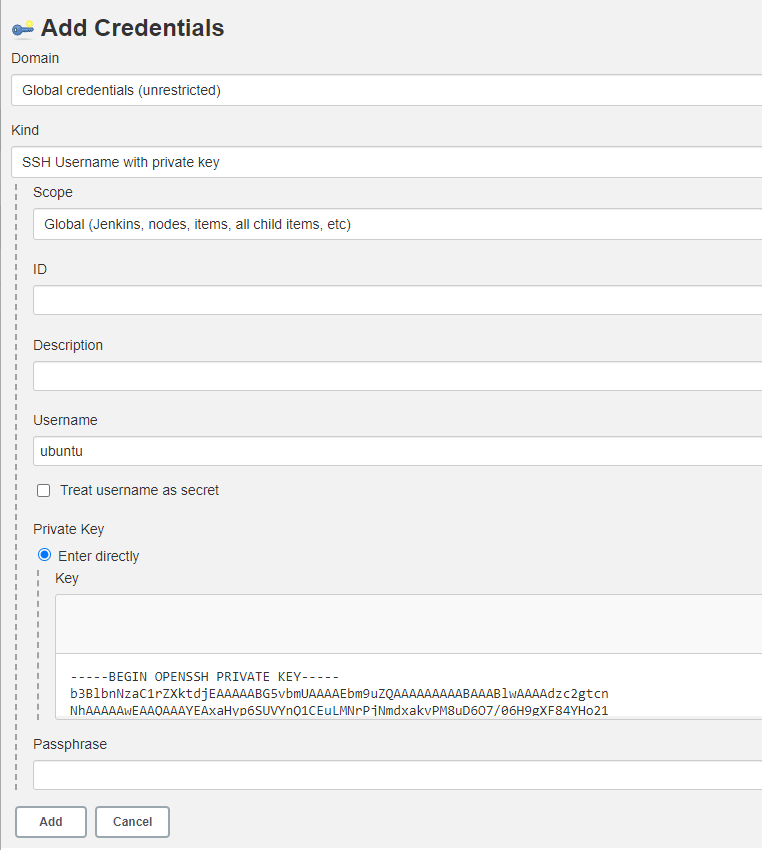
Give the node a description, set the remote root directory to: /home/ubuntu and give it a label.



Select the “Launch method” as “Launch agents via SSH”.

Enter the remote Slave node's private IP.

Press the “Add” Credentials button.



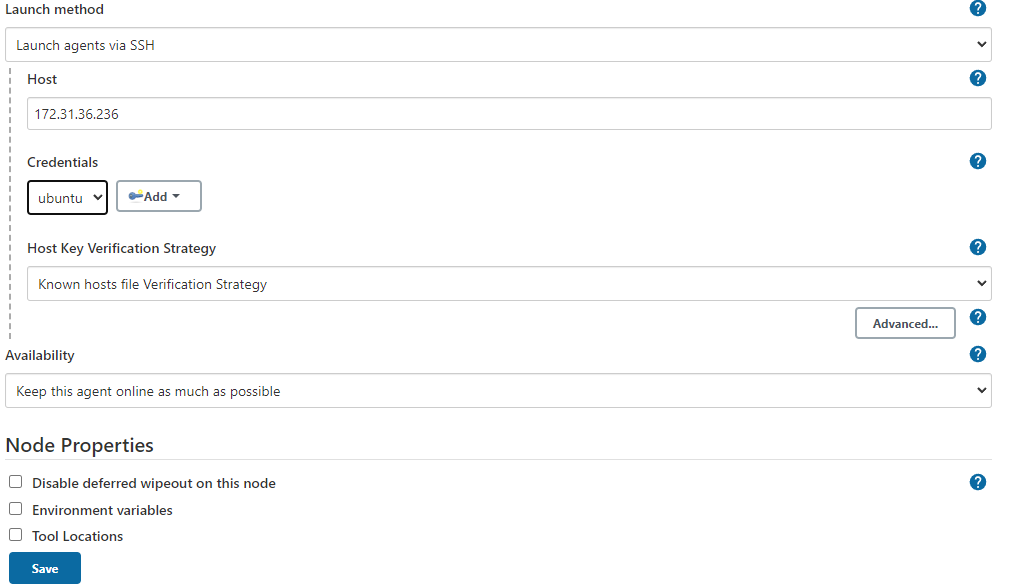
Select the Kind as “SSH Username & private key”.

“Username” is ubuntu.

Select the “Private Key, Enter Directly radio button”.

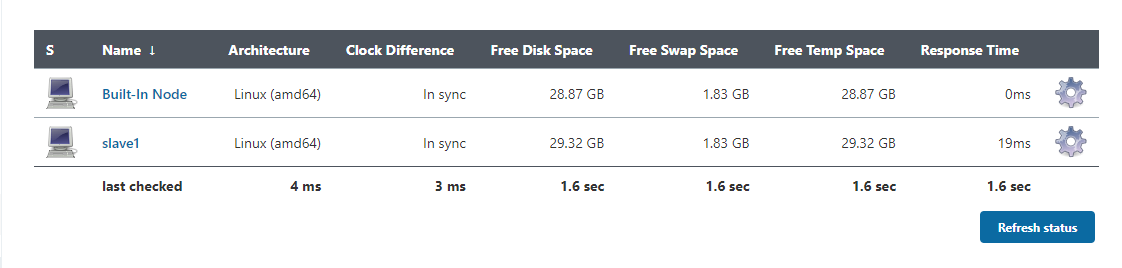
Paste in the private key that was copied out in the earlier stage.

Press the “Add” button.



Select this newly created credentials and press the save button.

Confirm the node has been added and is running by pressing the refresh status button.

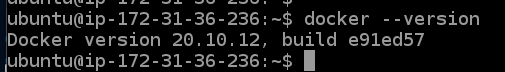


Here we can see both slaves have been created and are ready.

## Install Docker onto the Jenkins Slave

Run the following commands on the Jenkins Slave:

* sudo apt update
* sudo apt-get install ca-certificates curl gnupg lsb-release
* sudo apt-get update
* curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
* echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb\_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
* sudo apt-get update
* sudo apt-get install docker-ce docker-ce-cli containerd.io
* docker --version



Docker is now installed and ready to use on the Jenkins Slave.

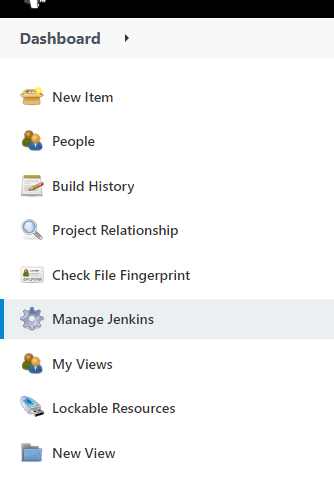
# Create Jenkins build pipeline

Now it's time to create the pipeline that will:

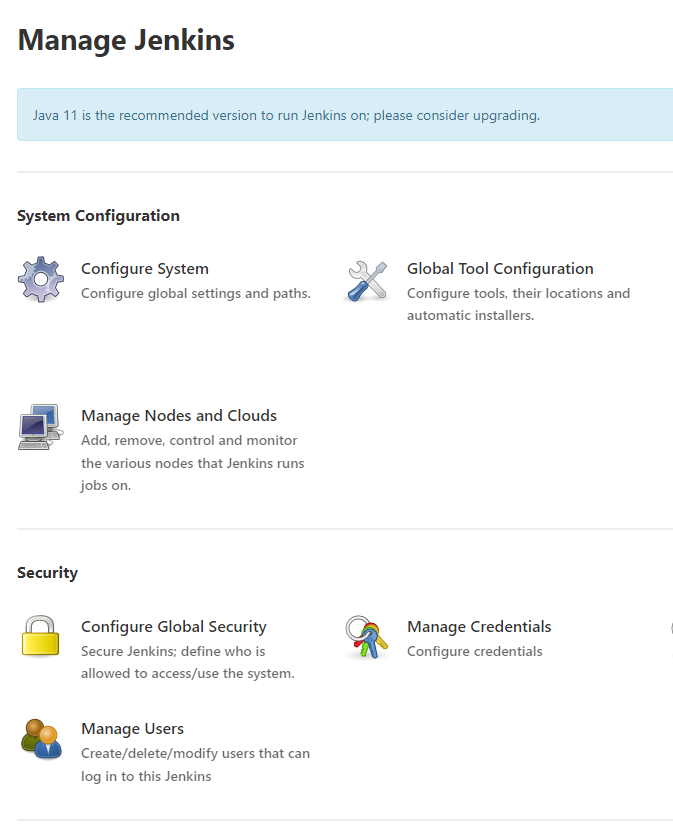
* Detect if the code has been released
* Build the code
* Unit test the code
* Package the code into a container
* Function test the container
* Deliver the code on success to private container repository.

## Add dockerhub access token to Jenkins

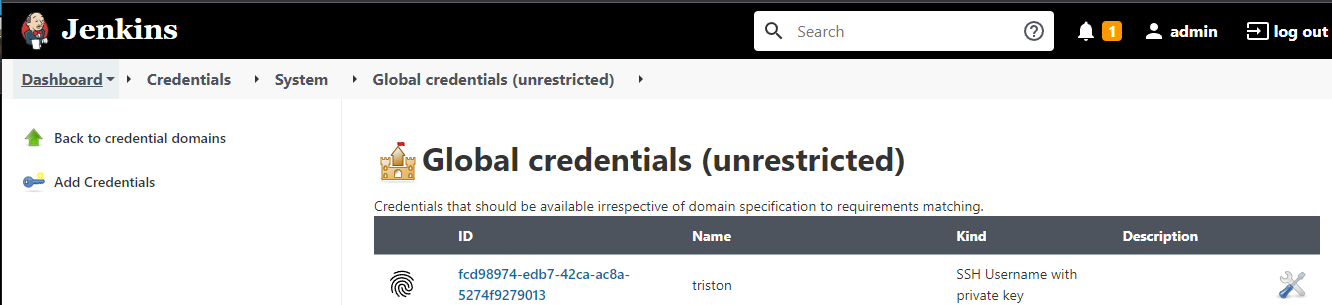
To allow the pipeline to continuously deliver the docker image to docker hub we will need to provide the dockerhub credentials to the pipeline.



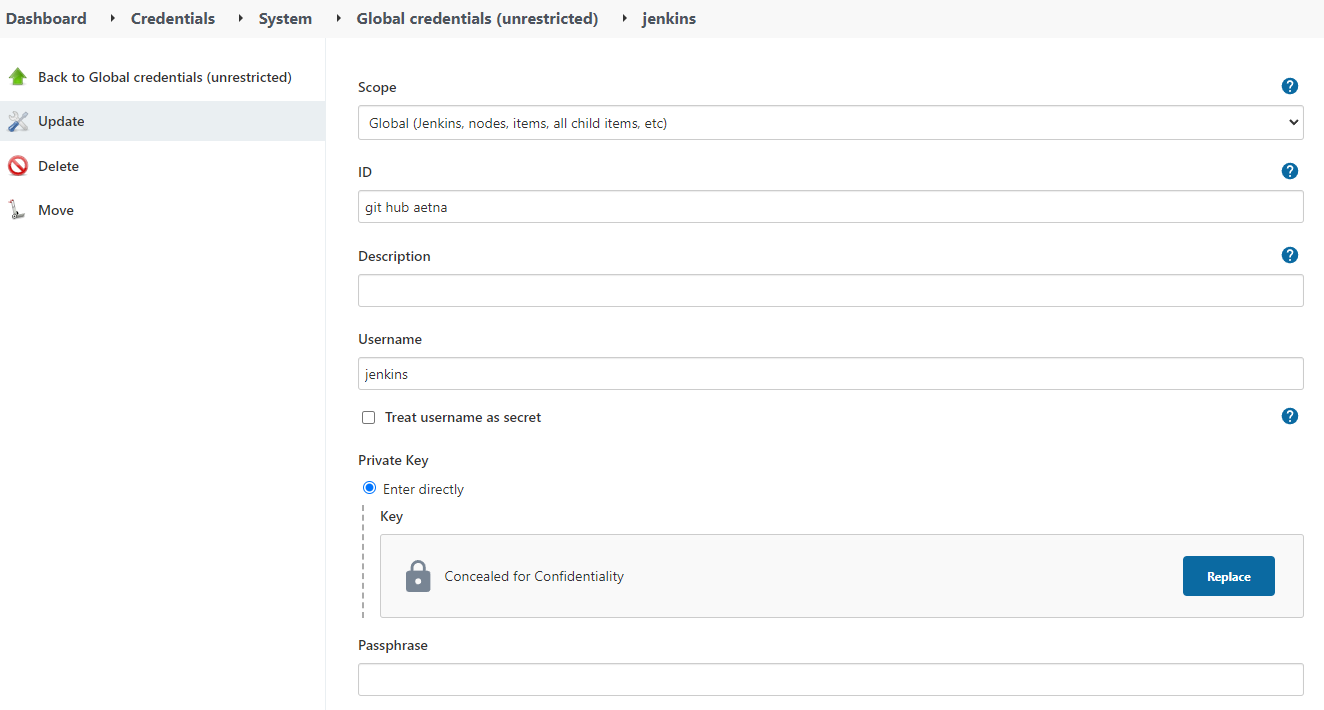
Press the “Manage Jankins” button



Press the “Manage Credentials” button.



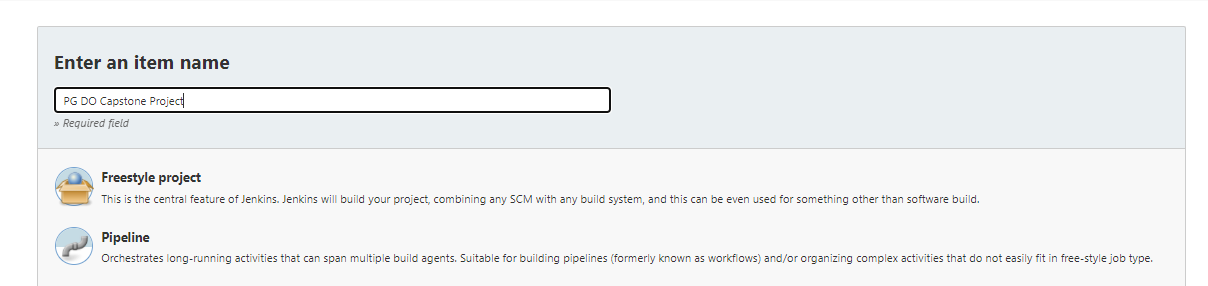
Select “System, Global credentials (unrestricted)” and press the “Add Credentails” button.



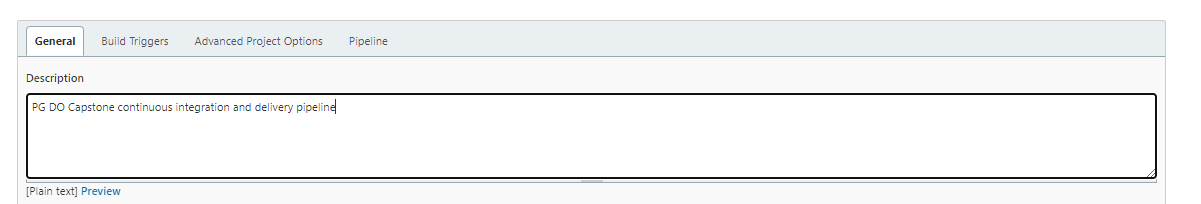
Enter the above details and enter the dockerhub access token created earlier into the private key section. Save and apply the changes.

## Configure the Build Pipeline

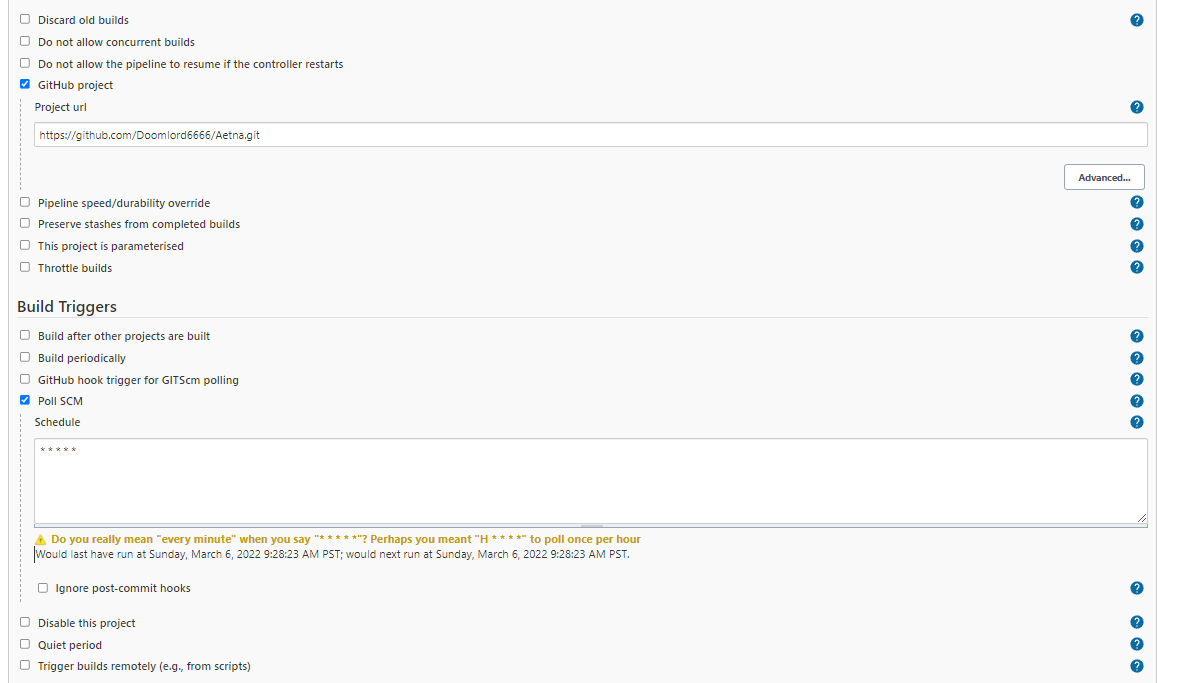
Log into Jenkins on the Jenkins Dashboard press the “New Item” icon.



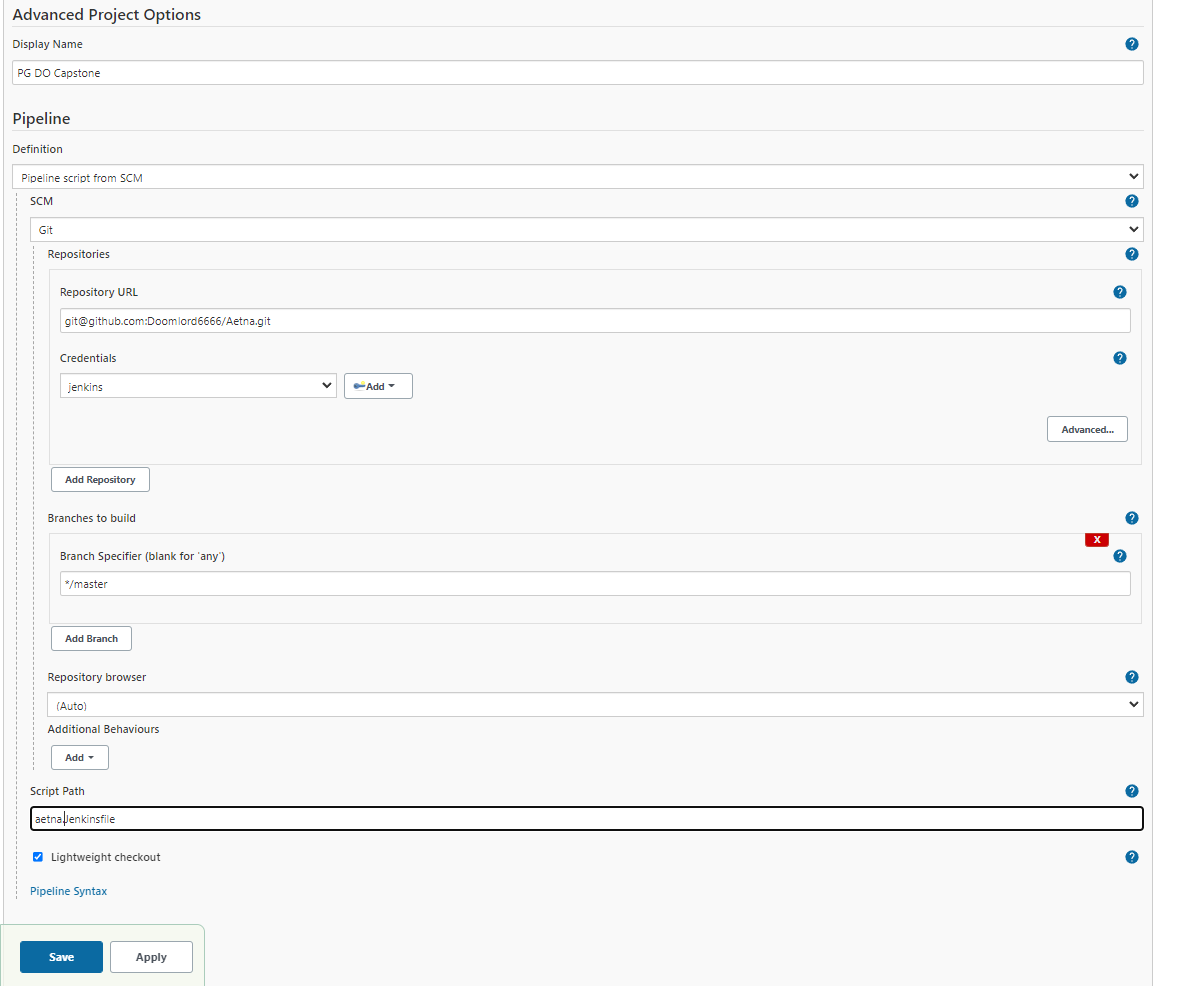
Press the “Pipeline” Tab to take us to the configuration.



Select the “General” tab and enter a description of the pipeline.



Set the Github project URL, trigger the pipeline to poll the Github for changes and set the interval to every minuite.



Configure Jenkins to look for the pipeline Jenkins file in the source code. This file contains all the stages that Jenkins should execute to complete the pipeline.

Press the “Save Button”.

# aetna.jenkinsfile

The Jenkins file contains all the stages that needs to be executed by Jenkins to complete the pipeline and contains the following code:

pipeline {

agent { label 'slave1' }

tools {

maven 'MyMaven'

jdk 'MyJava'

}

environment {

DOCKERHUB\_CREDENTIALS=credentials('docker\_hub')

}

stages {

stage ('Checkout the code') {

steps {

git branch: 'master',

credentialsId: 'git hub aetna',

url: 'git@github.com:Doomlord6666/Aetna.git'

}

}

stage ('Build the code') {

steps {

sh """

echo Compiling...

mvn compile

"""

}

}

stage ('UNIT Test') {

steps {

sh """

echo Unit Testing...

mvn test

"""

}

}

stage ('Package the code'){

steps{

sh"""

echo Packaging...

mvn package

"""

}

}

stage ('Create Docker image'){

steps{

sh"""

echo Building Image...

docker build -t doomlord6666/aetna:${BUILD\_NUMBER} .

"""

}

}

stage ('Perform Docker run'){

steps{

sh"""

echo Running Application...

docker run -d -p 8080:8080 --name bsafe\_${BUILD\_NUMBER} doomlord6666/aetna:${BUILD\_NUMBER}

"""

}

}

stage ('Function Test'){

steps{

sh(returnStdout: true, script: '''#!/bin/bash

sleep 10

HTTP=$(curl localhost:8080 --silent)

if [ "$HTTP" == "Greetings from aetna healthcare!" ]; then

echo test passed

exit 0

else

echo test failed

exit 1

fi

'''.stripIndent())

}

}

stage ('Push image to image repo'){

steps{

sh"""

echo $DOCKERHUB\_CREDENTIALS\_PSW | docker login -u $DOCKERHUB\_CREDENTIALS\_USR --password-stdin

docker push doomlord6666/aetna:${BUILD\_NUMBER}

docker tag doomlord6666/aetna:${BUILD\_NUMBER} doomlord6666/aetna:latest

docker push doomlord6666/aetna:latest

docker logout

"""

}

}

stage ('clean up'){

steps{

sh"""

docker stop bsafe\_${BUILD\_NUMBER}

docker rm bsafe\_${BUILD\_NUMBER}

docker image rm doomlord6666/aetna:${BUILD\_NUMBER}

docker image rm doomlord6666/aetna:latest

rm ${WORKSPACE}/\* -f -r

"""

}

}

}

}

Press the “Save” button.

## Code Breakdown

This Instructs Jenkins to only use Jenkins or agents that are labeled up as “slave1” for this pipeline.

pipeline {

agent { label 'slave1' }

This instructs Jenkins we will be using Maven and Java JDK in the pipeline as defined in the Jenkins Global configuration and to run the job on the jenkins slave.

tools {

maven 'MyMaven'

jdk 'MyJava'

}

This instruction tells jenkins to create a variable called DOCKERHUB\_CREDENTIALS and populate it with the docker hub credentials created ealier.

environment {

DOCKERHUB\_CREDENTIALS=credentials('docker\_hub')

}

Now we move onto the individual stages that make up the pipeline. The stage checks out the code from git hub.

stage ('Checkout the code') {

steps {

git branch: 'master',

credentialsId: 'git hub aetna',

url: 'git@github.com:Doomlord6666/Aetna.git'

}

This stage builds the code using maven.

stage ('Build the code') {

steps {

sh """

echo Compiling...

mvn compile

"""

}

This stage runs the maven unit tests against the built code.

stage ('UNIT Test') {

steps {

sh """

echo Unit Testing...

mvn test

"""

}

This stage packages the code using maven

stage ('Package the code'){

steps{

sh"""

echo Packaging...

mvn package

"""

}

This stage builds a docker image from a docker file and the code with a unique identifier. It also provides the remote private docker registry details.

stage ('Create Docker image'){

steps{

sh"""

echo Building Image...

docker build -t doomlord6666/aetna:${BUILD\_NUMBER} .

"""

}

This stage creates and runs a docker container exposed on port 8080 from the docker image.

stage ('Perform Docker run'){

steps{

sh"""

echo Running Application...

docker run -d -p 8080:8080 --name bsafe\_${BUILD\_NUMBER} doomlord6666/aetna:${BUILD\_NUMBER}

"""

}

This stage runs a function test against the running container

stage ('Function Test'){

steps{

sh(returnStdout: true, script: '''#!/bin/bash

sleep 10

HTTP=$(curl localhost:8080 --silent)

if [ "$HTTP" == "Greetings from aetna healthcare!" ]; then

echo test passed

exit 0

else

echo test failed

exit 1

fi

'''.stripIndent())

}

}

This stage pushes and tags the docker image into the remote private registry. It creates both a latest and build numbered version.

stage ('Push image to image repo'){

steps{

sh"""

echo $DOCKERHUB\_CREDENTIALS\_PSW | docker login -u $DOCKERHUB\_CREDENTIALS\_USR --password-stdin

docker push doomlord6666/aetna:${BUILD\_NUMBER}

docker tag doomlord6666/aetna:${BUILD\_NUMBER} doomlord6666/aetna:latest

docker push doomlord6666/aetna:latest

docker logout

"""

}

}

This stage stops the running docker container, removes it, removes the docker image and finally the Jenkins workspace.

stage ('clean up'){

steps{

sh"""

docker stop bsafe\_${BUILD\_NUMBER}

docker rm bsafe\_${BUILD\_NUMBER}

docker image rm doomlord6666/aetna:${BUILD\_NUMBER}

docker image rm doomlord6666/aetna:latest

rm ${WORKSPACE}/\* -f -r

"""

}

# Dockerfile

The dockerfile resides in SCM and is used to generate the docker image from the code and looks like this:

FROM ubuntu:latest

RUN apt-get update && apt-get install -y openjdk-8-jdk

ENV JAVA\_HOME=/usr/lib/jvm/java-8-openjdk-amd64

ENV PATH=$PATH:$JAVA\_HOME/bin

WORKDIR /tmp

COPY target/ .

ENTRYPOINT ["java","-jar","/tmp/bsafe-0.0.1-SNAPSHOT.jar"]

## Code Breakdown

Download the latest ubuntu docker image to use as the base

FROM ubuntu:latest

Update the image and install Java 8

RUN apt-get update && apt-get install -y openjdk-8-jdk

Set the Java environment variables

ENV JAVA\_HOME=/usr/lib/jvm/java-8-openjdk-amd64

ENV PATH=$PATH:$JAVA\_HOME/bin

Set the docker working directory to /tmp

WORKDIR /tmp

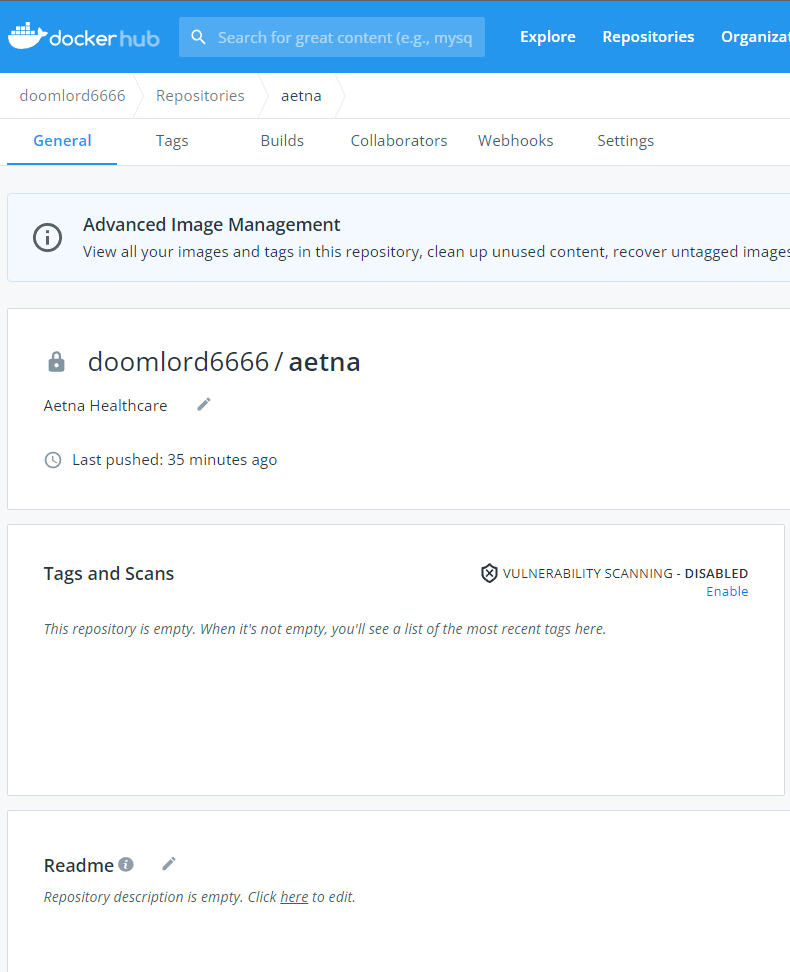
Copy the built application code into the working directory

COPY target/ .

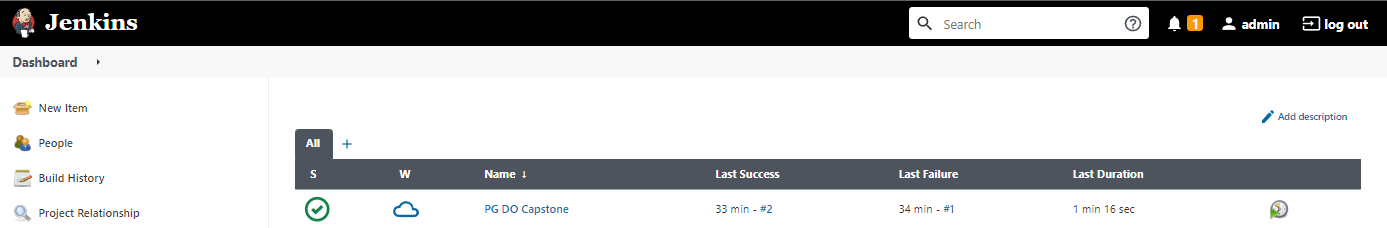
Set the entry point into the docker container

ENTRYPOINT ["java","-jar","/tmp/bsafe-0.0.1-SNAPSHOT.jar"]

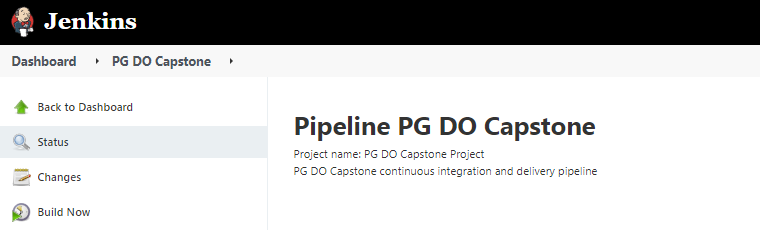
# Running and checking the pipeline.



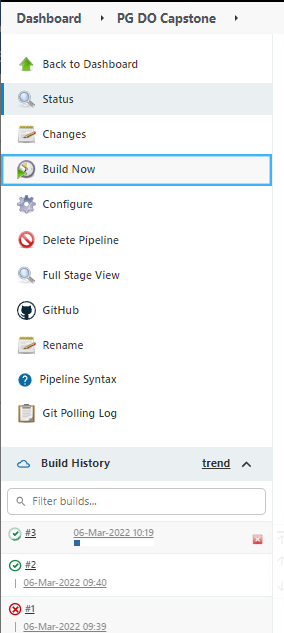
Firstly, we check dockerhub and we can see there are no docker images.



On the Jenkins dashboard select the “PG DO Capstone” pipeline

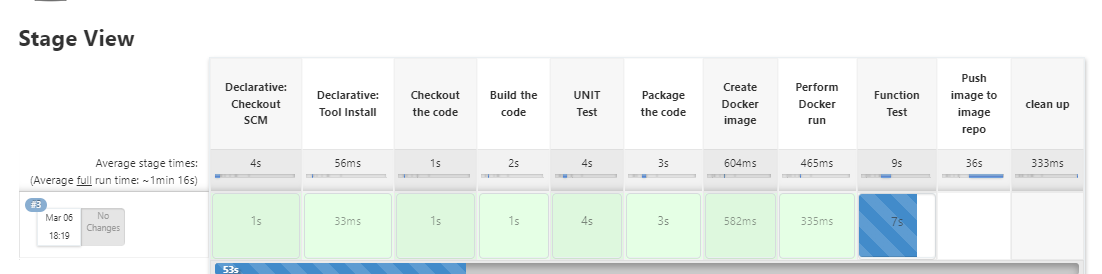


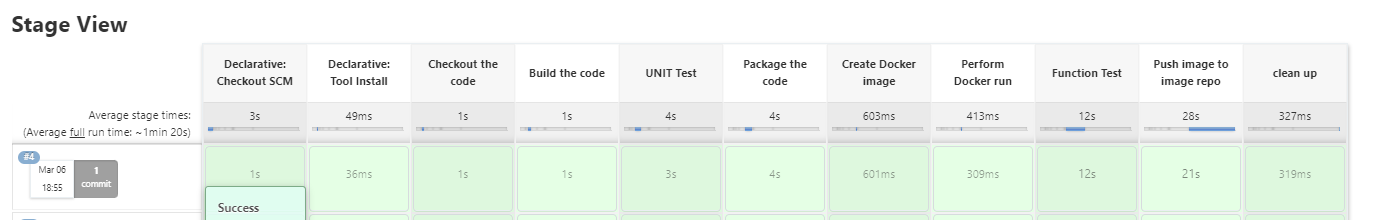
Press the “Build Now” button.



In the left-hand corner, we can see the build is running and is running.

If we click on the pipeline, we can get a stage view with stages that match up to our Jenkins file.



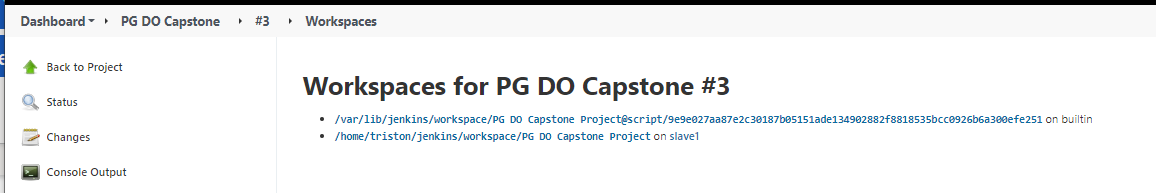


Here we can see that all stages completed and how long they took.

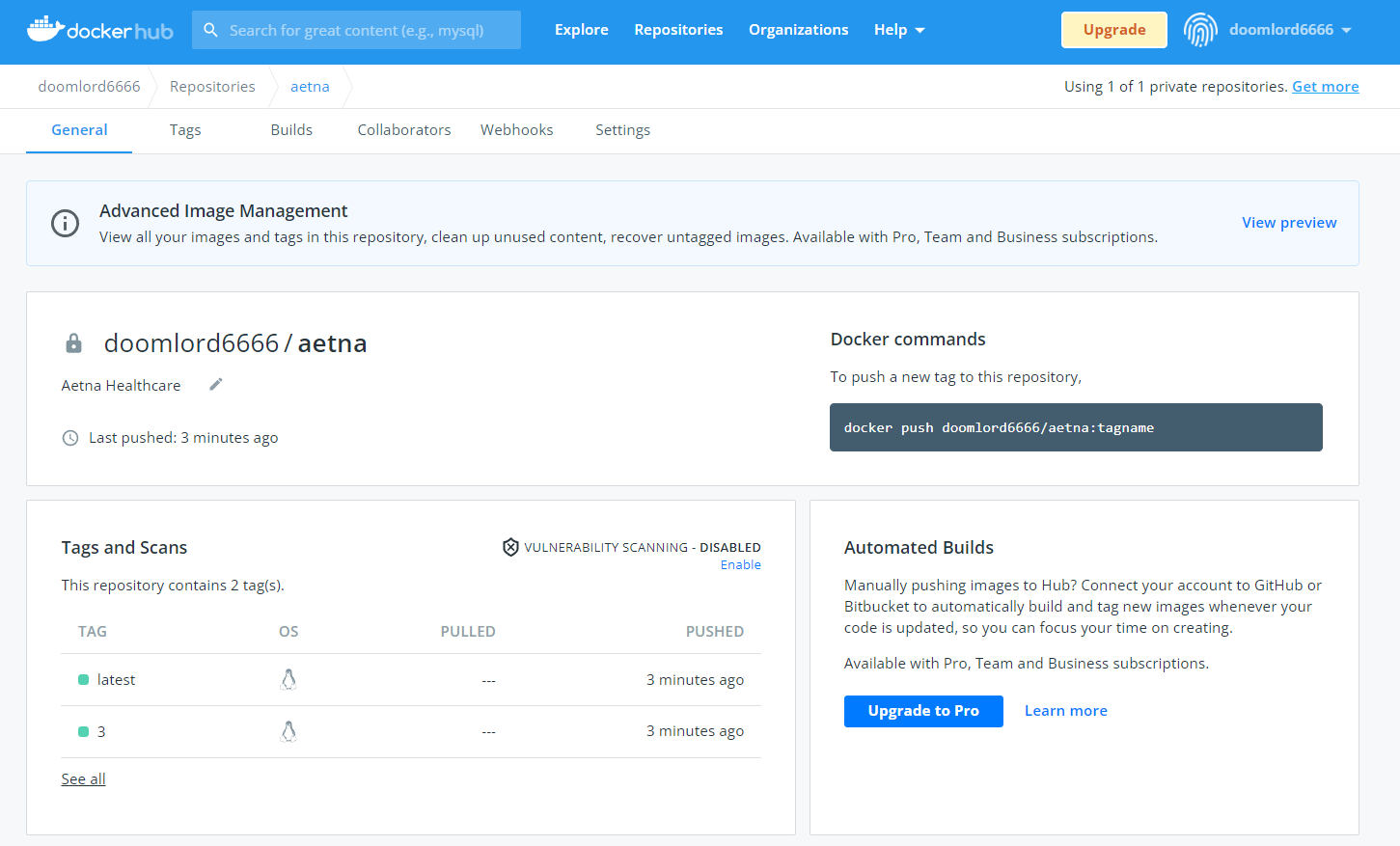
If we select the last build, #3



Press the “Workspaces” icon.



We can see the job ran on slave1



And we can see the docker images have been created and placed into dockerhub and the build number ties up to the Jenkins build number.

Conclusion

In conclusion I achieved the following requirements:

* Install and configure the Jenkins architecture on AWS instance
* Use the required plugins to run the build creation on a containerized platform
* Create and run the Docker image which will have the application artifacts
* Execute the automated tests on the created build
* Create your private repository and push the Docker image into the repository
* Expose the application on the respective ports so that the user can access the deployed application
* Remove container stack after completing the job

# Enhancements

There are a number of enhancements that could be made:

* Improved notifications via email or slack.
* Improved tagging – Mark the images with the commit code and also git tag the code when a release is moved into production.
* Improved error handling on the pipeline script.
* Use a multibranch pipeline script to allow for parallel unit, integration, function and regression testing.
* Change the SCM poll to a Webhook.
* The use of multiple slaves to help with load.
* The use of the Jenkins docker image instead of running.

# USP

* Faster deployment cycles
* More predictable and reliable testing
* Faster bug fixes
* Measurable progress
* Better code quality
* Consistent application and infrastructure builds
* repeatability
* Less risk
* Happier customers