Jenkins

Course Details:

1. What is Jenkins
2. Difference between Continuous Integration, continuous delivery and continuous deployment
3. Installation and configuration
4. Plugins
5. Security Setup for Users
6. Global Tool Configuration
7. Jobs/Project in Jenkins
8. GitHub, Maven Integration
9. Parameter Builds
10. Job Triggers/Scheduling
11. Publish over SSH
12. Mail Notifications
13. Master Slave architecture
14. Pipeline through Upstream and Downstream Jobs
15. Jenkins Pipeline:
    - Declarative
    - Scripted
16. End to end CICD demo with git, maven, nexus, sonar, tomcat
17. Multibranch pipeline
18. Blue ocean
19. Jenkins CLI
20. Extra Preparation for Jenkinsfile

Cloudjunction

1. What is Jenkins?

Jenkins is a self-contained, open source automation server which can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software.

Why Jenkins?

1. Jenkins is an **open source** automation tool written in Java with plugins built for Continuous Integration purpose.
2. **Plugins** allows the integration of Various DevOps stages. If you want to integrate a tool, you need to install the plugins for that tool. For example: Git, Maven project, Amazon EC2, Ansible etc.
3. Ease of use: easy **web interface**

Cloudjunction

2. Difference between Continuous Integration, continuous delivery and continuous deployment

Continuous Integration

Continuous Integration (CI) is the process of automating the build and testing of code every time a team member commits changes to version control. CI encourages developers to share their code and unit tests by merging their changes into a shared version control repository after every small task completion. Committing code triggers an automated build system to grab the latest code from the shared repository and to build, test, and validate the full master branch (also known as the trunk or main).

Continuous Delivery

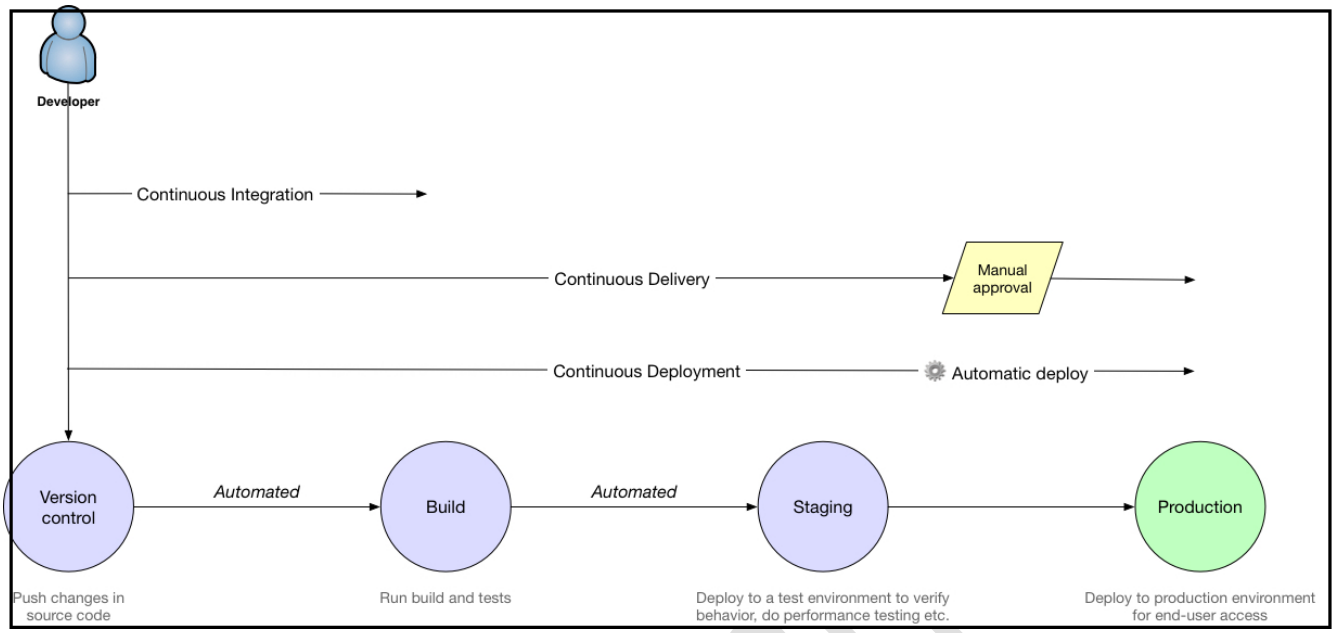
Continuous Delivery (CD) is the process to build, test, configure and deploy from a build to a production environment. Multiple testing or staging environments create a *Release Pipeline* to automate the creation of infrastructure and deployment of applications. Successive environments support progressively longer-running activities of integration, load, and user acceptance testing.

Continuous delivery is attractive because it automates the steps between checking code into the repository and deciding on whether to release well-tested, functional builds to your production infrastructure.

Continuous Deployments

Continuous deployment is an extension of continuous delivery that automatically deploys each build that passes the full test cycle. Instead of waiting for a human gatekeeper to decide what and when to deploy to production, a continuous deployment system deploys everything that has successfully traversed the deployment pipeline.

Continuous deployment also allows organizations to benefit from consistent early feedback.



Cloudjunction

3. Installation Steps on Ubuntu 18:

Open Jdk 11 installation

apt update

apt install openjdk-11-jdk

Install Jenkins

Add the repository key to the system.

wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -

Append the Debian package repository address to the server's sources.list

sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'

apt update

apt install jenkins

service jenkins start

What does this package do?

- The 'jenkins' user is created to run this service.
- Log file will be placed in /var/log/jenkins/jenkins.log. Check this file if you are troubleshooting Jenkins.
- By default, Jenkins listen on port 8080. Access this port with your browser to start configuration.

If you want to change the port, edit the /etc/default/jenkins to replace the line

HTTP_PORT=8080

by

HTTP_PORT=8090 (Here, 8090 was chosen but you can put another port available)

Setting up Jenkins

To set up our installation, we'll visit Jenkins on its default port, 8080, using the server domain name or IP address: http://ip_address_or_domain_name:8080

We should see "*Unlock Jenkins*" screen, which displays the location of the initial password


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

copy the 32-character alphanumeric password from the terminal and paste it into the "Administrator password" field, then click "Continue".

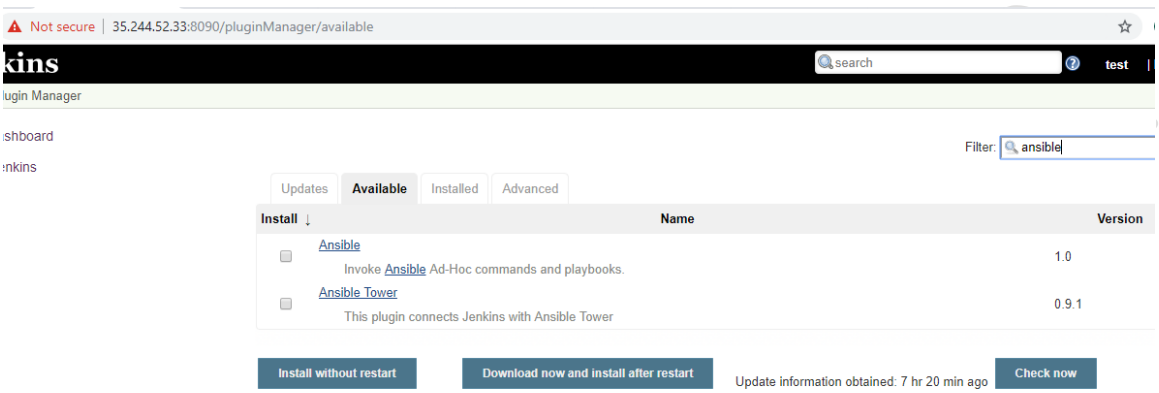
Install suggested plugins.

4. Jenkins Plugins

Plugins are the primary means of enhancing the functionality of a Jenkins environment to suit organization- or user-specific need

Go to Manage Jenkins-->Manage plugins

Here you can check plugin details and install appropriate plugins



Commonly used plugins

Github

EC2

Green balls

Ansible

Nexus

Sonar

pipeline

5. Security Setup for Users

Immediately after installation, Jenkins will allow anyone to run anything as user jenkins, which is bad

The Configure Global Security page has two sections in which you:

1. **Security realm:**

Determine who is allowed access and establish the user authentication method here

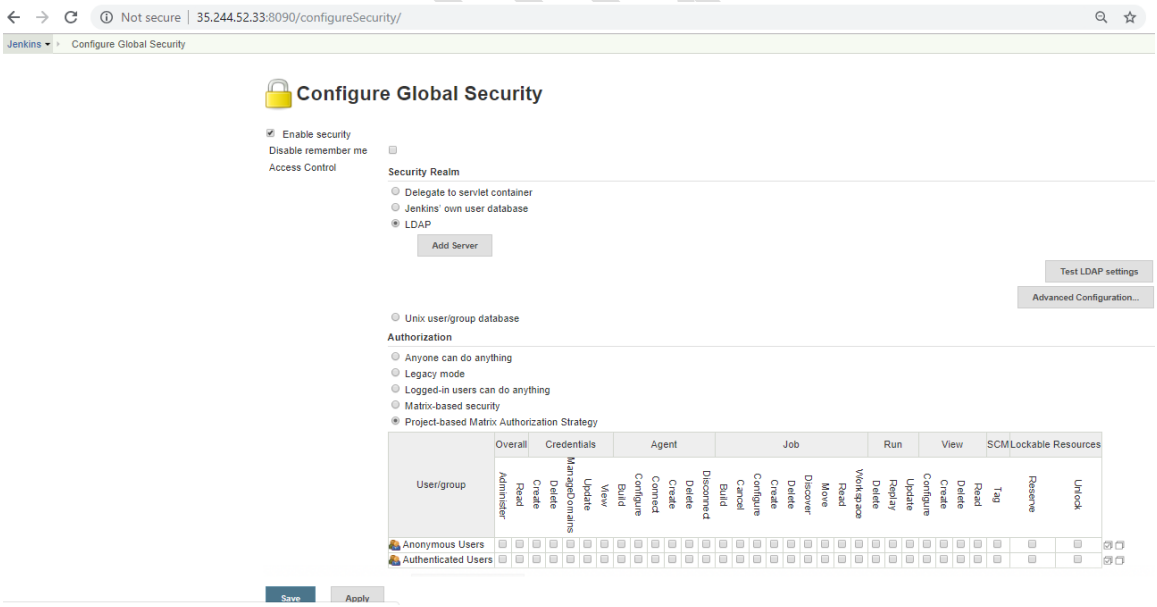
- Jenkins' Own User Database: Jenkins maintains its own independent user database

- LDAP Similar to Active Directory

2. **Authorization:**

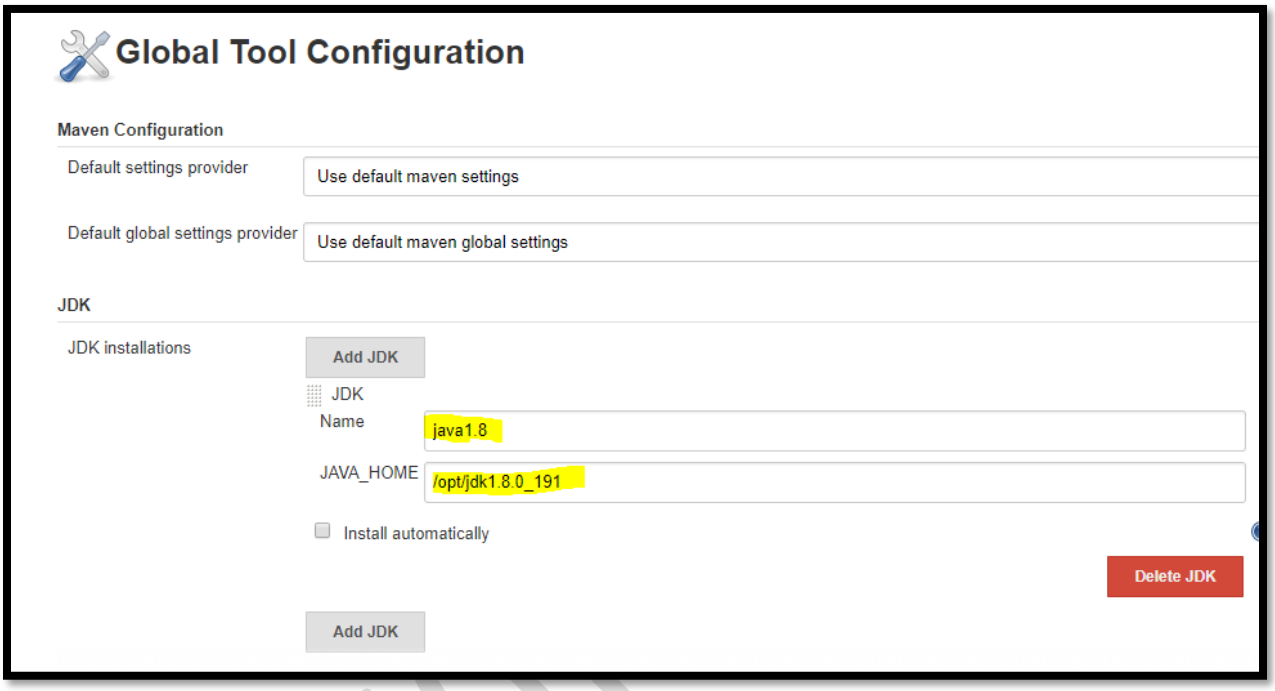
allows you to configure what users are allowed to do once authenticated

- Matrix-based Security



6. Global Tool Configuration

configure various tools that we need to utilize at the time of creating a build job, for example, Java, Maven.



7. Jobs in Jenkins

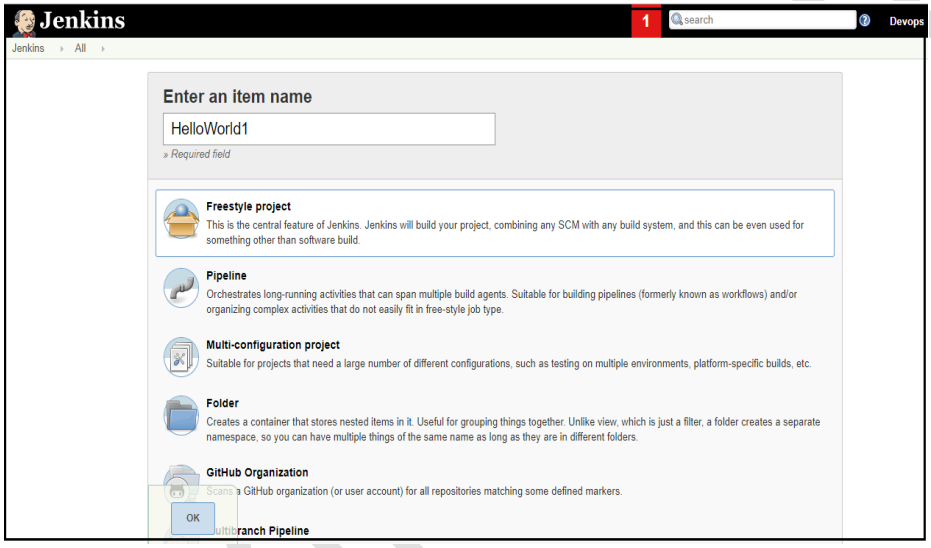
Jenkins build job as a particular task or step in your build process

Login the Jenkins application

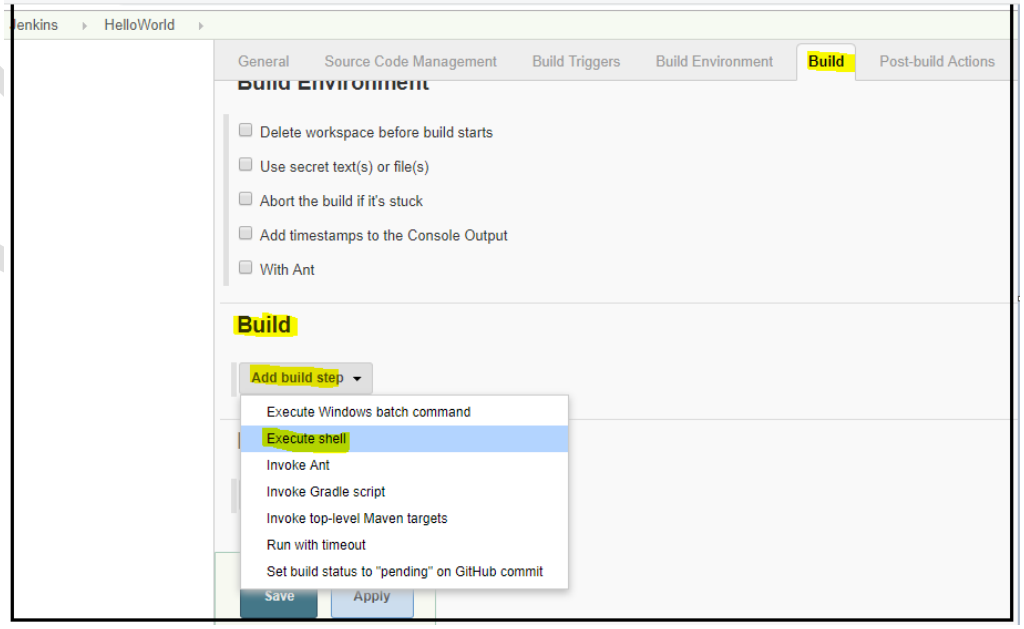
http://<Public_IP>:8080

Provide credentials created in installation steps

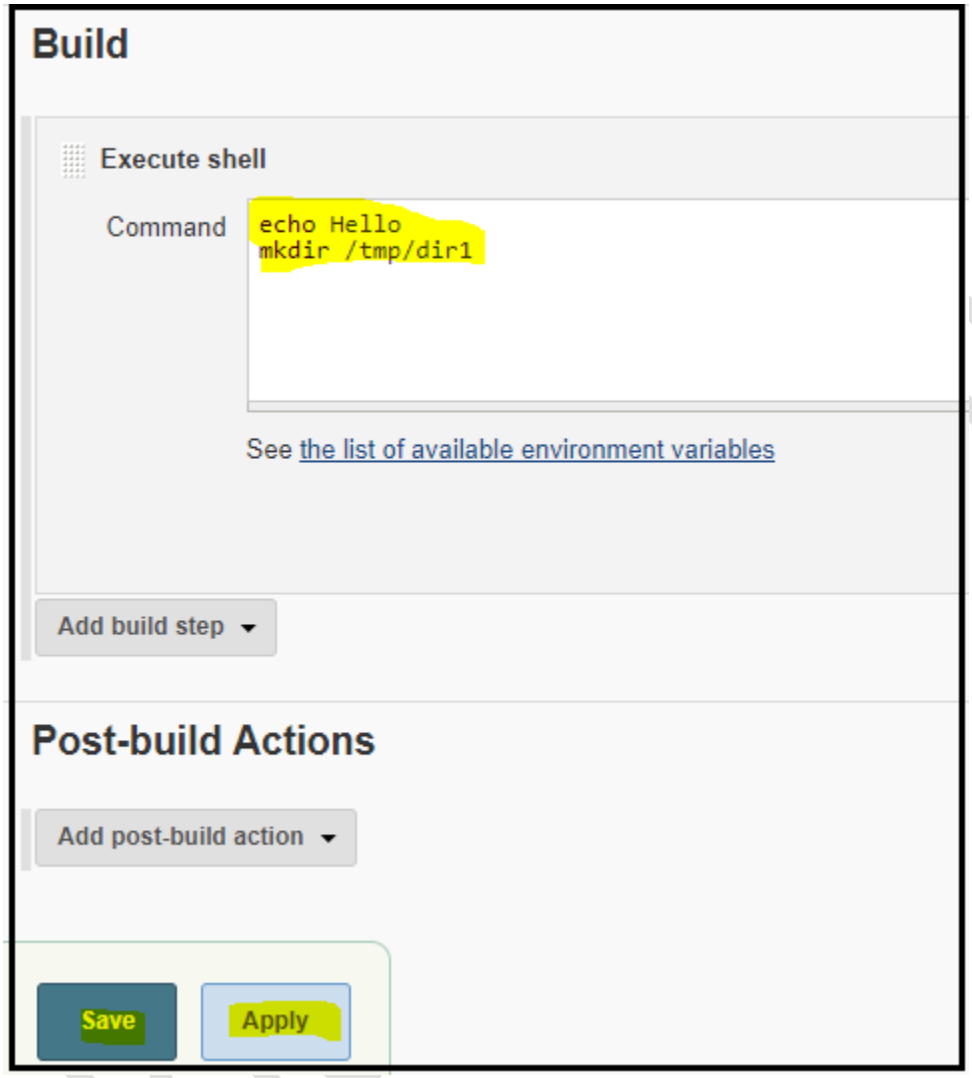
1. Click on New item → provide the project name→select freestyle project

a. 

2. Click Build→Add Build steps→Execute shell



3. Enter below script



4. click on Build Now.

Sometimes we get error “sudo: no tty present and no askpass program specified”

Solution:

 # vi /etc/sudoers

add following line at the end. After ROOT Line

jenkins ALL= NOPASSWD: ALL

Setting timezone in server

#cat /etc/timezone

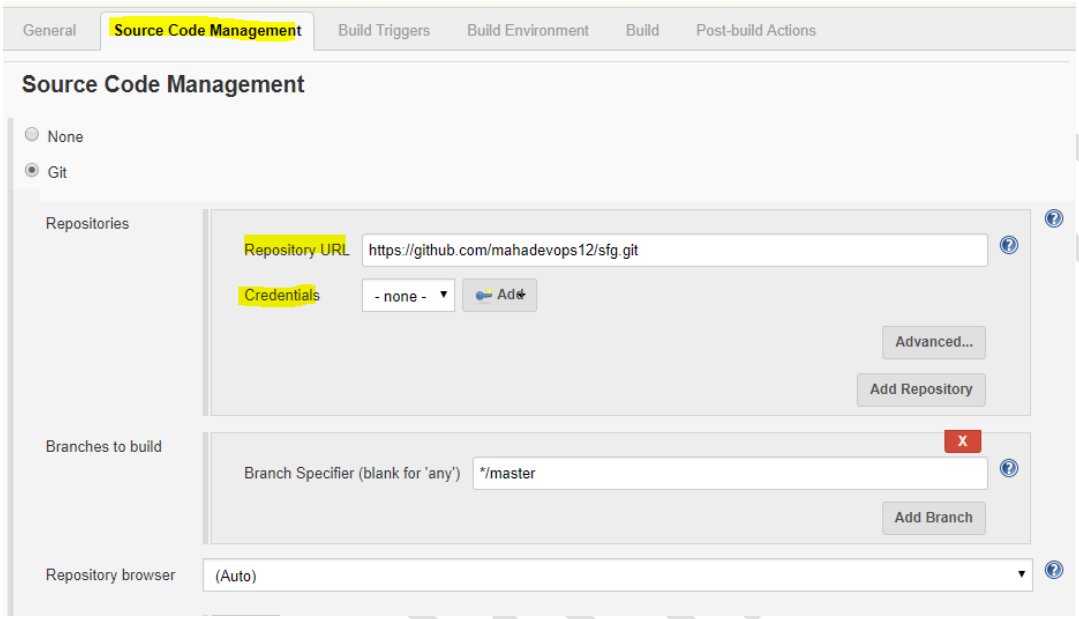
#timedatectl

#timedatectl set-timezone Asia/Kolkata

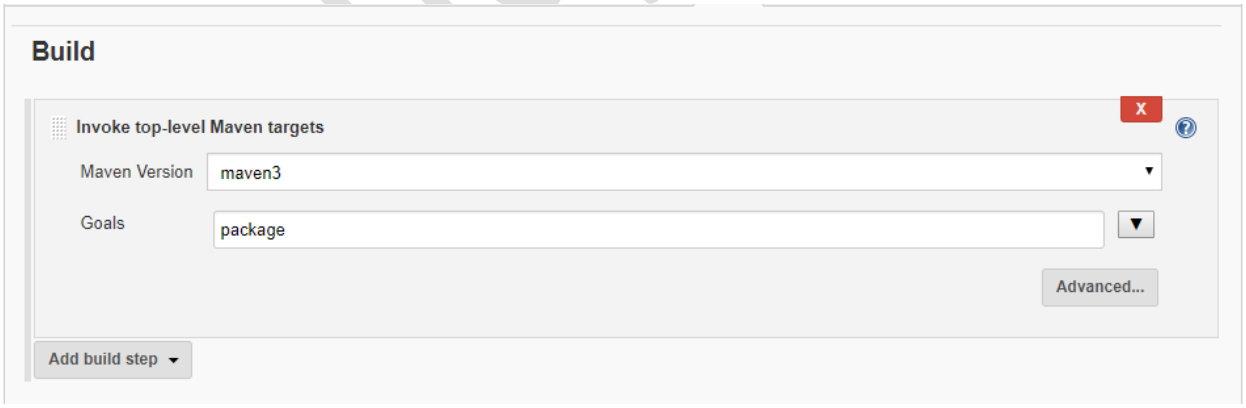
Cloudjunction

8. Github & Maven with Jenkins

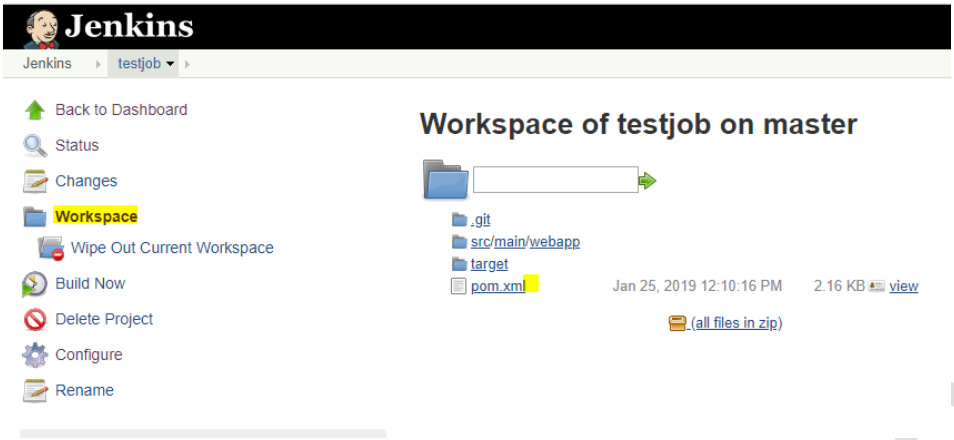
First check github plugin was there and if not, install it.



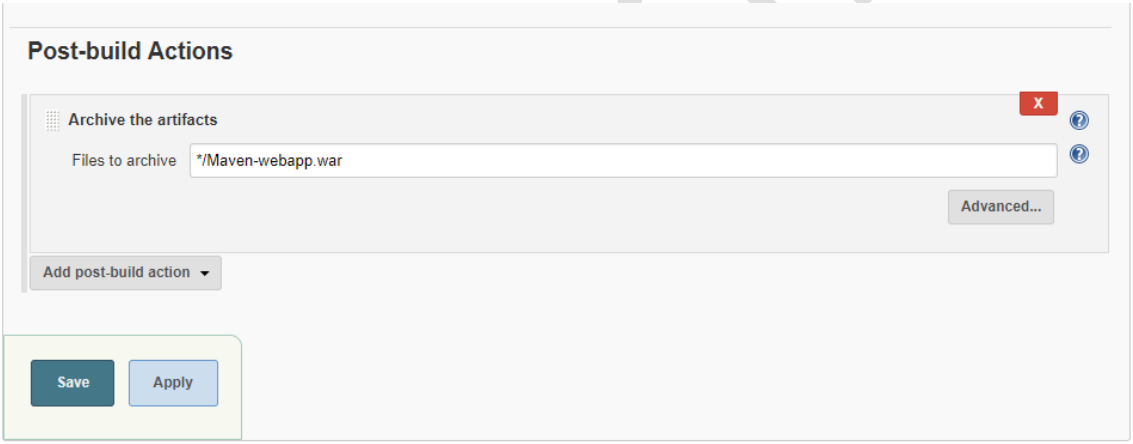
Maven integration



Managing Artifacts



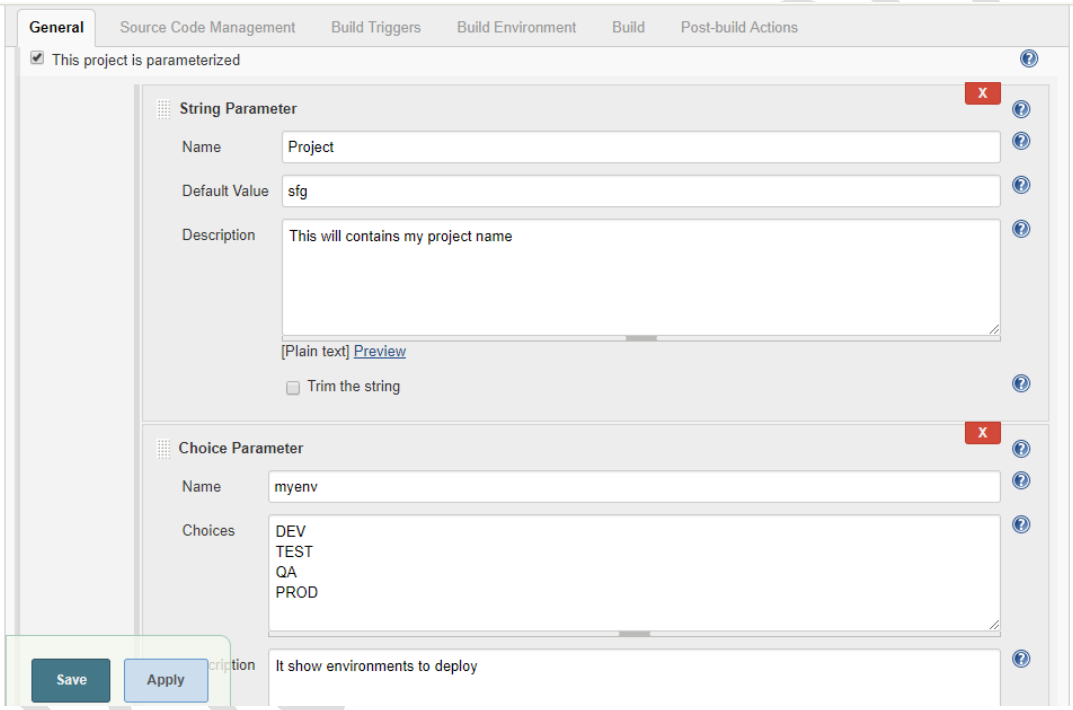
Archive artifacts



9. Parameters Build

Types of parameters

- String
- Choice
- Boolean
- file



You can access parameter value as $Pramater_Name

In above example you can access it $Project and $myenv

10. Job Triggers/Scheduling

Build periodically: Provides a cron-like feature to periodically execute this project.

Poll SCM: Configure Jenkins to poll changes in SCM.

Webhooks: Jenkins will receive PUSH hook from repository

Cron Syntax

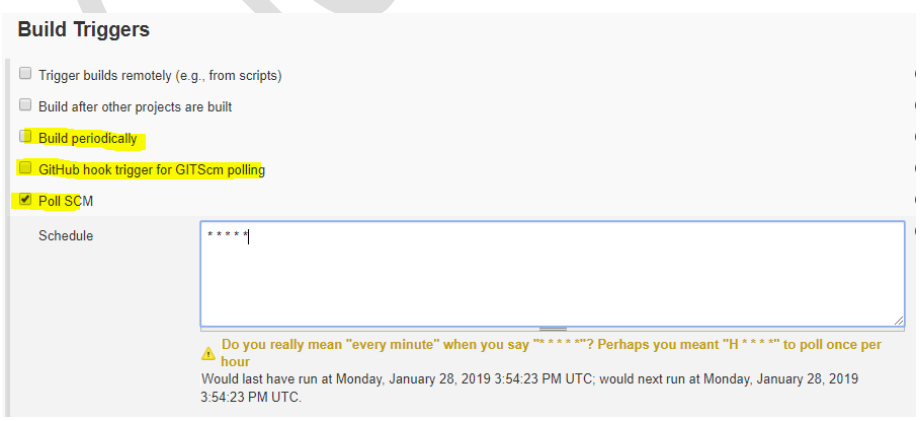
*	*	*	*	*
MINUTE	**HOUR**	**DOM**	**MONTH**	**DOW**
Minutes within the hour (0–59)	The hour of the day (0–23)	The day of the month (1–31)</td>	The month (1–12)	The day of the week (0–7) where 0 and 7 are Sunday.</td>

To specify multiple values for one field, the following operators are available. In the order of precedence,

- * specifies all valid values
- M-N specifies a range of values
- M-N/X or */X steps by intervals of X through the specified range or whole valid range
- A,B,…,Z enumerates multiple values

In addition below are supported as convenient aliases

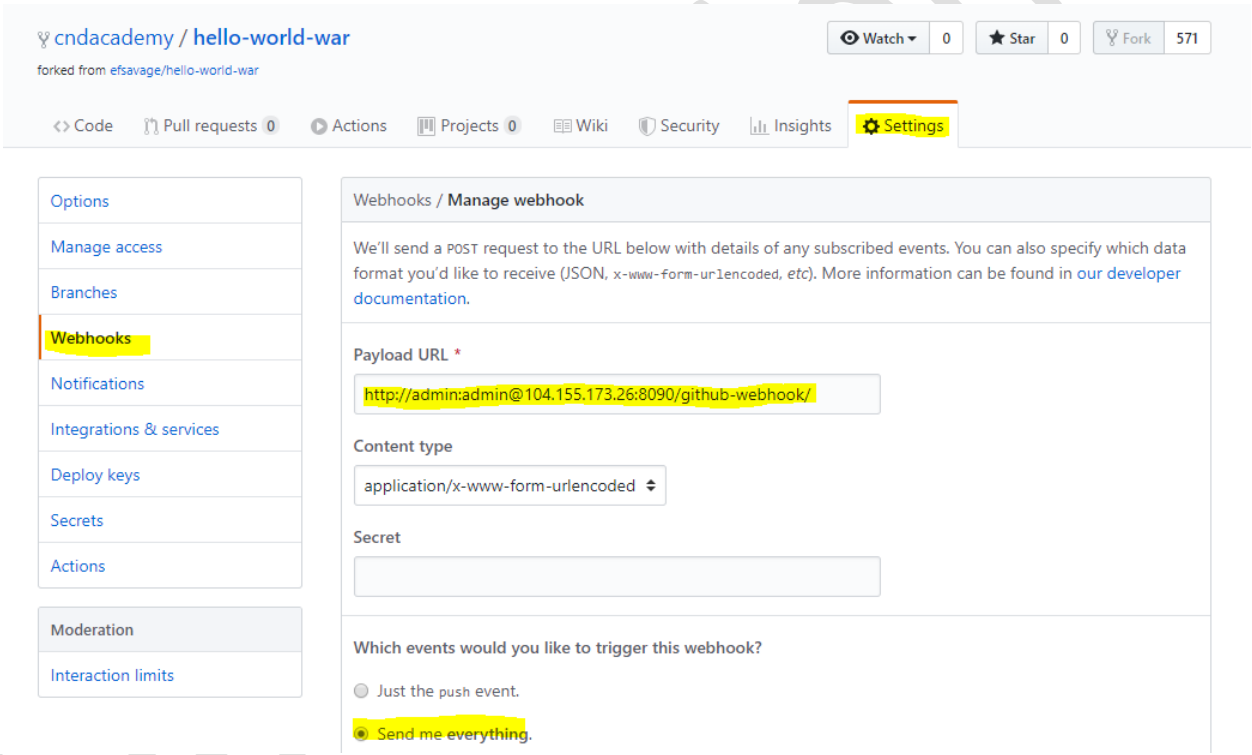
@yearly, @annually, @monthly, @weekly, @daily, @midnight, @hourly



Webhooks

Got to your Github Repository

1. Go to your project repository.
2. Go to "settings" in the right corner.
3. Click on "webhooks."
4. Click "Add webhooks."
5. Write the Payload URL as Eg **http://user:password@35.223.79.59:8080/github-webhook/**



Now go to the job's build trigger and select **GitHub hook trigger for GITScm polling**

For testing purpose commit to github repository will trigger the build automatically.

11. Publish Over SSH

SCP - Send files over SSH

This is used to copy files/artifacts from Jenkins to Remote server such dev.

On Master Jenkins

su jenkins

ssh-keygen

#ls ~/.ssh

#cat ~/.ssh/id_rsa.pub

On tomcat-dev instance [dev server]

#apt update

#apt install openjdk-11-jdk tomcat9 -y

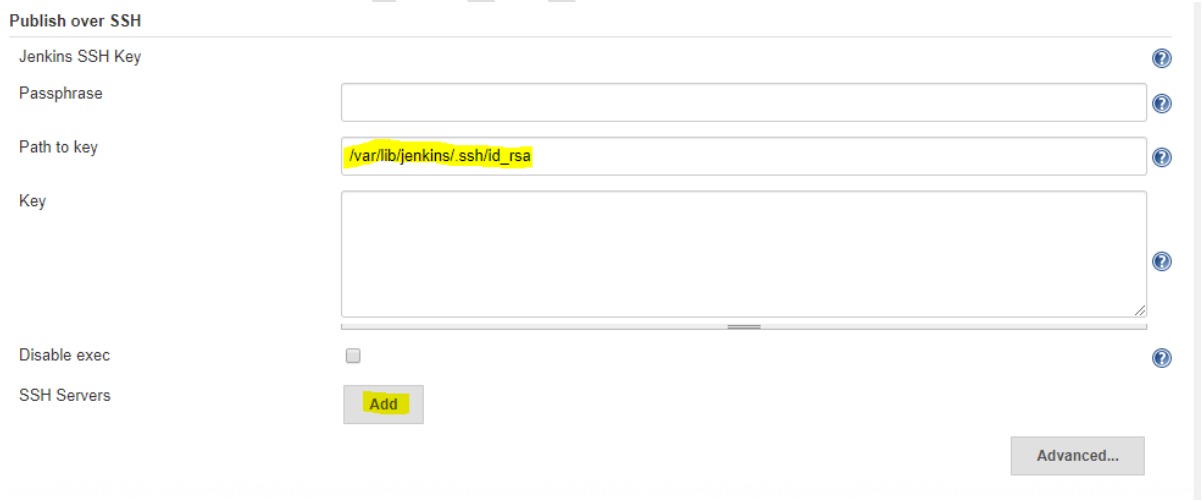
#vi ~/.ssh/authorized_keys

Here pastes public key from Jenkins master ~/.ssh/id_rsa.pub

On Jenkins UI

Install plugin "publish over ssh"

Manage Jenkins→Configure System→Publish over ssh



We will add tomcat dev instance as SSH server.

SSH Servers

SSH Server

Name

dev-server

Hostname

34.125.28.218

Username

root

Remote Directory

/var/lib/tomcat9/webapps

Advanced...

Success

Test Configuration

Delete

Now configure Jobs in Jenkins to copy file over SSH.

JOB→Build→Send files or execute command over ssh

Send files or execute commands over SSH

SSH Publishers

SSH Server

Name

dev-server

Advanced...

Transfers

Transfer Set

Source files

target/*.war

Remove prefix

target

Remote directory

Exec command

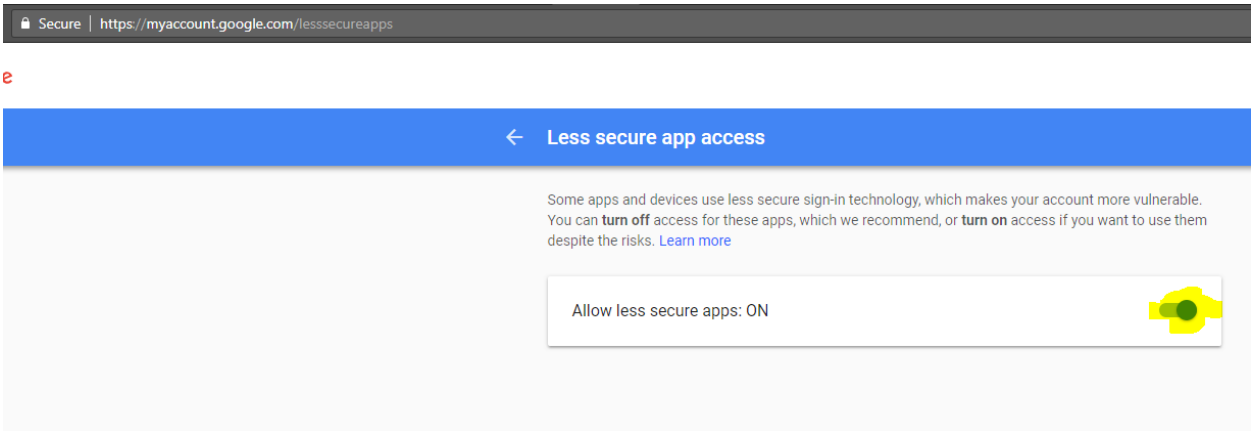
All of the transfer fields (except for Exec timeout) support substitution of Jenkins environment variables

Advanced...

12. Jenkins Mail Configurations

Please turn ON Allow less secure apps

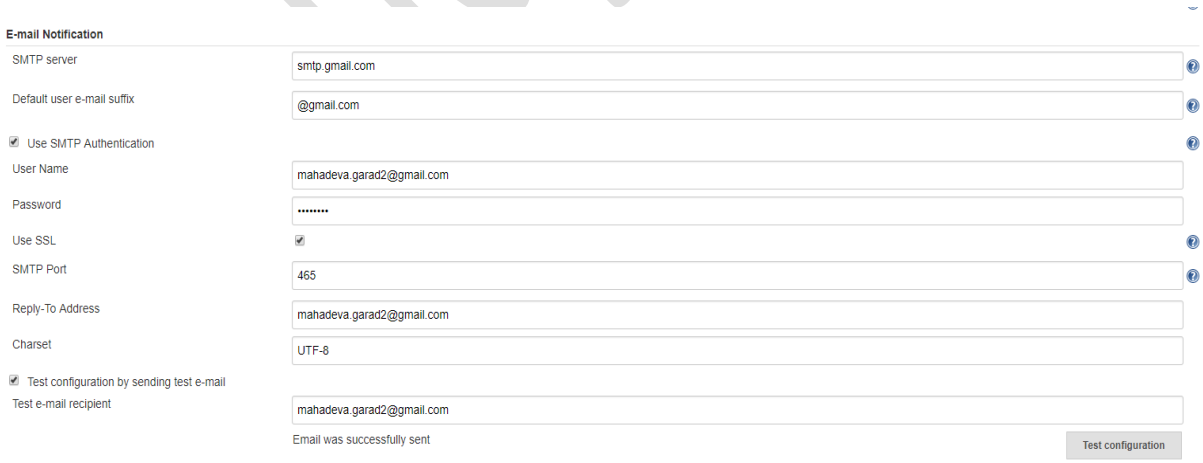
https://myaccount.google.com/lesssecureapps



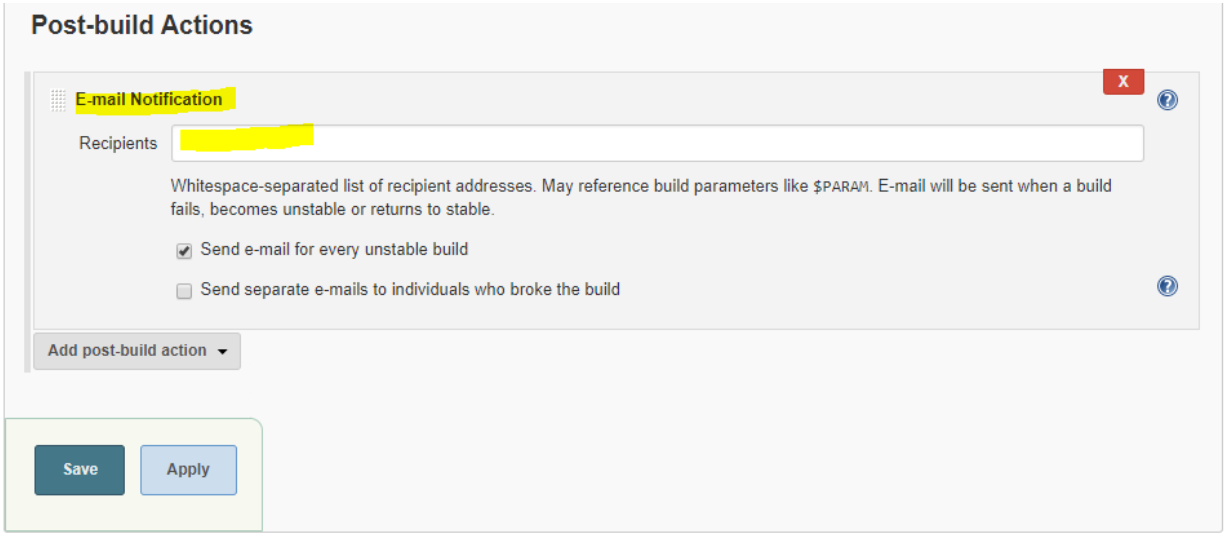
Also Create Google App password: https://support.google.com/accounts/answer/185833?hl=en

Now we will configure email notification on Jenkins

Manage Jenkins→Configure System



You can configure email notification in post build action



Cloudjum

13. Master Slave architecture

- Jenkins supports the master-slave architecture, i.e. many slaves work for a master. It is also known as Jenkins Distributed Builds.
- We can configure a project to always run on a Slave machine.
- The job of a Slave is to do as they are told to, which involves executing build jobs dispatched by the Master.

Step1. On Master

Generate ssh keys

#ssh-keygen

Step2. On slave

#passwd root (Change root password)

vi /etc/ssh/sshd_config (edit sshd config file)

Change below lines

PermitRootLogin yes

PasswordAuthentication yes

service ssh restart (Restart service to impact changed configuration)

apt install openjdk-11-jdk (Install jdk)

create one directory for jenkins functions

Step3. On Master

#ssh-copy-id root@<private_ip_of_slave> (to copy public ip of master to slave)

ssh <private_ip_slave> (to verify connection with slave)

Step4. On Jenkins GUI

Manage Jenkins→Manage nodes→New Node

Cloudjunction

Configure node

Name: node

Description:

# of executors: 2

Remote root directory: /root/jenkins

Labels: slave1

Usage: Only build jobs with label expressions matching this node

Launch method: Launch agent agents via SSH

Host: 10.142.0.5

Credentials: - none - Add

The selected credentials cannot be found

Host Key Verification Strategy: Non verifying Verification Strategy

Advanced...

Jenkins Credentials Provider: Jenkins

Add Credentials

Domain: Global credentials (unrestricted)

Kind: SSH Username with private key

Scope: Global (Jenkins, nodes, items, all child items, etc)

Username: root

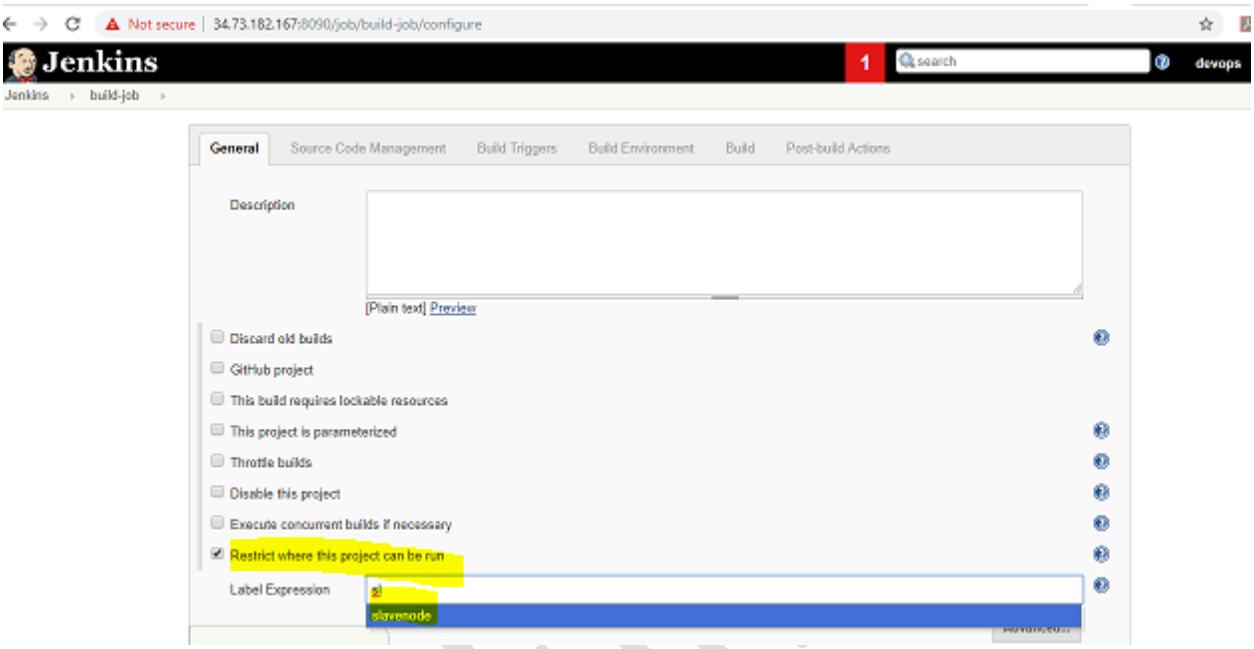
Private Key: Enter directly

Key:


```
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEAsAj2crT5yyQZU6Il0A8uxOGngYMtVupvLVc38PPRLmKRiMkp
aaxrLmjQydBxEkkeWhClOm/XIXfiGrfW1pnKuJnvLRQMrbrmwnnJW0lrr+sUJb9C
E5OMChzAQ3ch7XOQ7qsH3D8qbLM6yzdyufqbGKS/fPLVV7t0c3OX4eLj0gHTEPXf
eZcGoVVcVnyDZPC0thUU3C3i/aB139xxn/q4FePpSQB5IB+ApJ7ZRjQAd8TaTrc8
```


Passphrase:

Once slave node is setup, you can restrict a job where to run.



Cloudjun

14. Upstream/Downstream Jobs through Build Pipeline Plugin

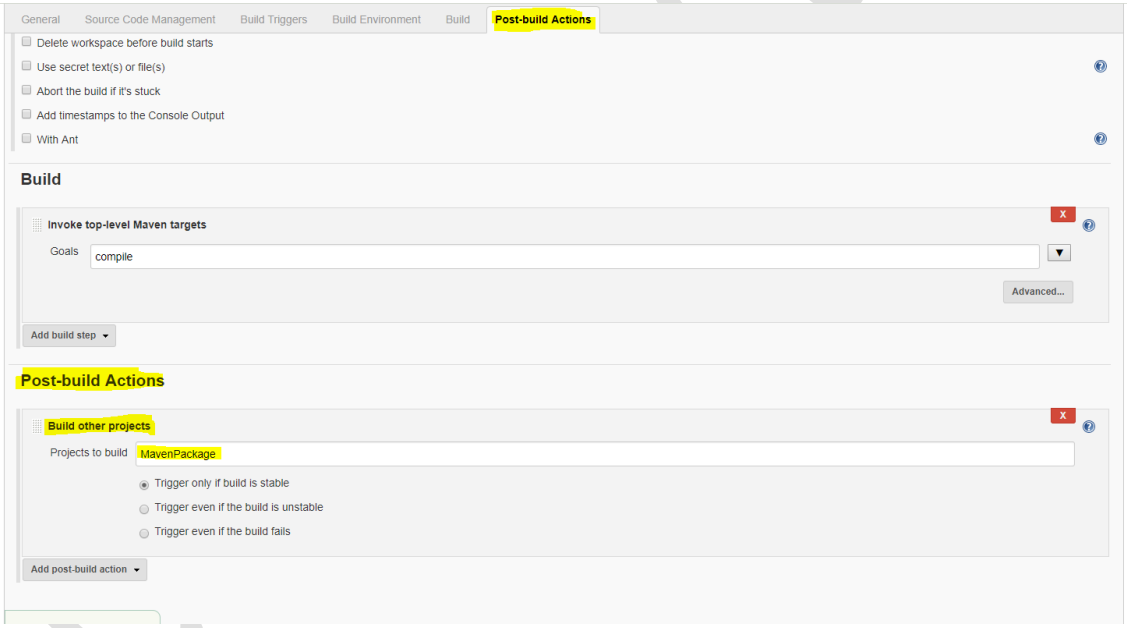
MavenCompile : Build Action →Invoke top level maven target→enter clean goal

MavenTest: Build Action →Invoke top level maven target→enter test goal

MavenPackage: Build Action →Invoke top level maven target→enter package goal

Please enter github repository in source code management for all jobs.

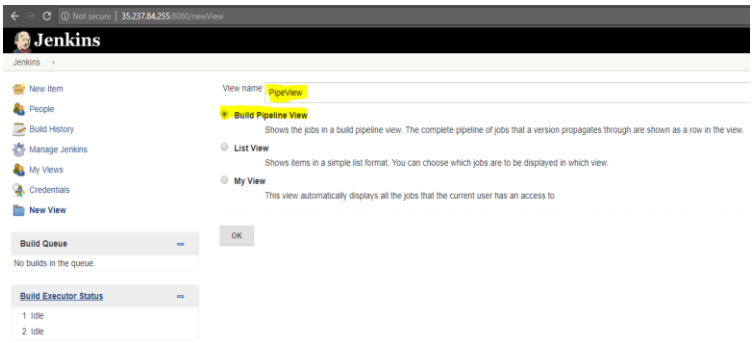
In MavenCompile job select → Post-build Actions → Build Oher Projects → Specify next job to executed (MavenTest)

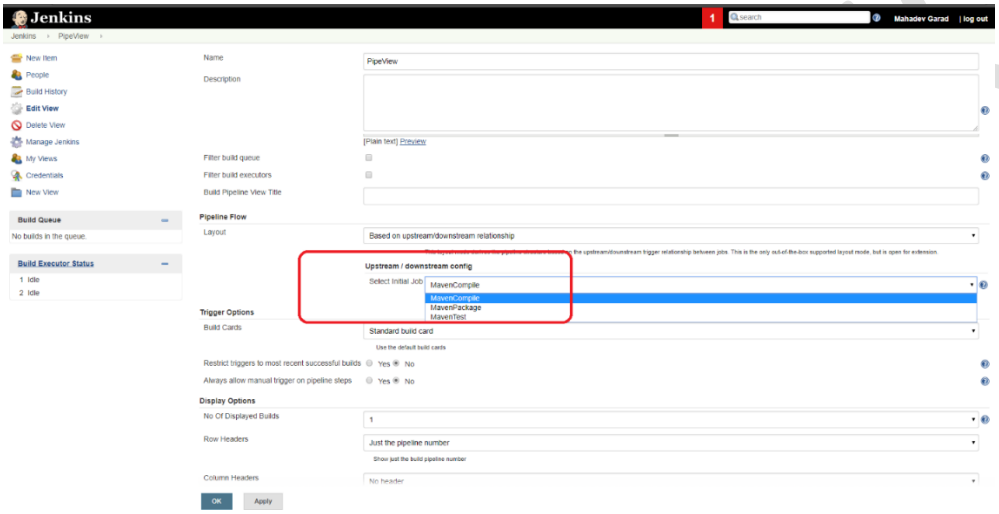


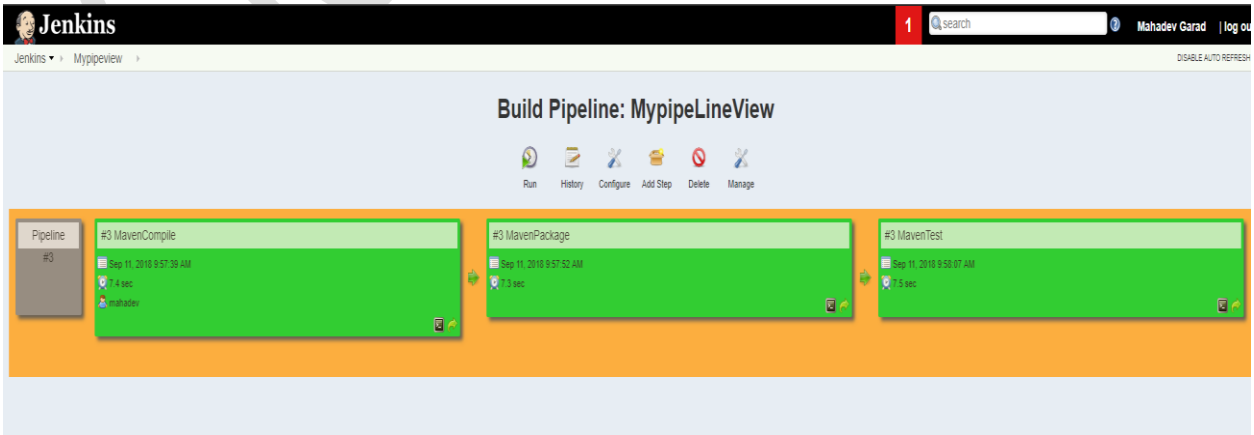
Similarly for MavenTest project select → Post-build Actions → Build Oher Projects → Specify next job to executed (MavenPackage)

Install build pipeline plugin

Now Click on + (New view) on Jenkins home to create to new view

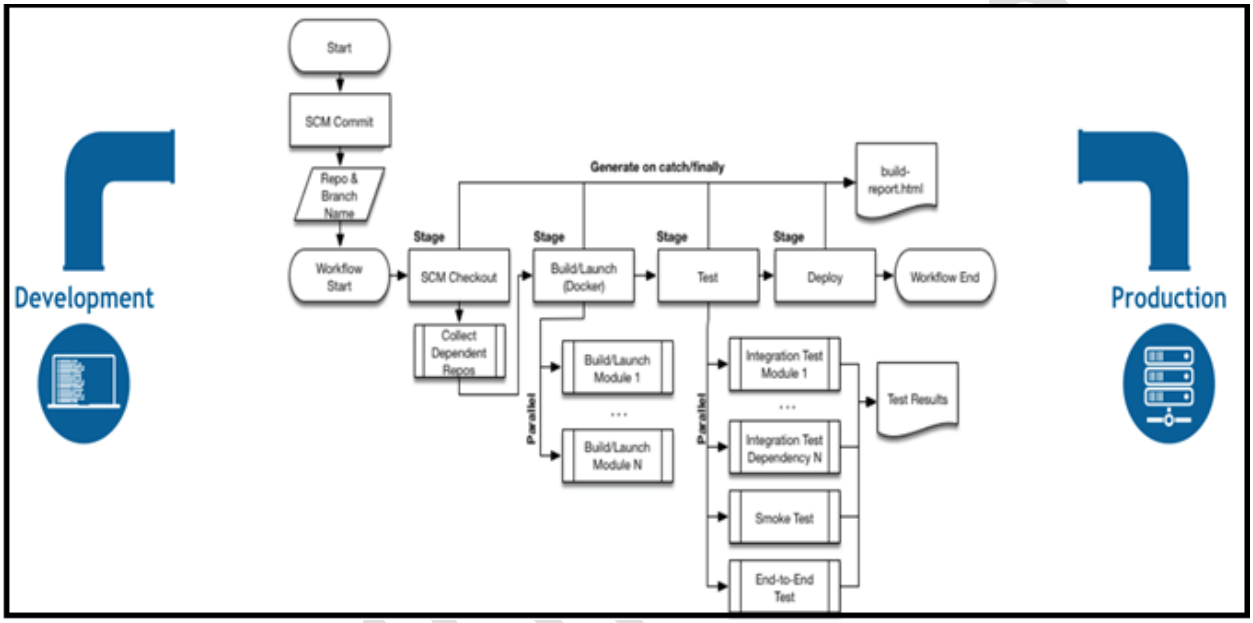






15. Jenkins Pipeline (Pipeline-as-code)

Jenkins Pipeline (or simply "Pipeline" with a capital "P") is a suite of plugins which supports implementing and integrating *continuous delivery pipelines* into Jenkins.



For writing code install Visual Studio Code.

Install Plugin

- Jenkins Doc
- Jenkinsfile support

Jenkinsfile: It is a text file that contains the definition of a Jenkins Pipeline and is checked into source control.

Pipeline supports two types of syntaxes:

1. **Declarative** :

   recent feature of Jenkins Pipeline which provides richer syntactical features over Scripted Pipeline syntax, and is designed to make writing and reading Pipeline code easier.

Example1

```
pipeline {
   agent any

   stages {
      stage('Build') {
         steps {
            echo 'Building..'
         }
      }
      stage('Test') {
         steps {
            echo 'Testing..'
         }
      }
      stage('Deploy') {
         steps {
            echo 'Deploying....'
         }
      }
   }
}
```

Cloudjunction

Example2 Scripted

```
node {
   stage('Build') {
         echo 'Building..'
   }
   stage('Test') {
            echo 'Testing..'

   }
   stage('Deploy') {
             echo 'Deploying....'
   }
}
```

Cloudjunction

16. End to end CICD demo with git,maven,nexus,sonar,tomcat

Jenkins Intergration With Sonarqube

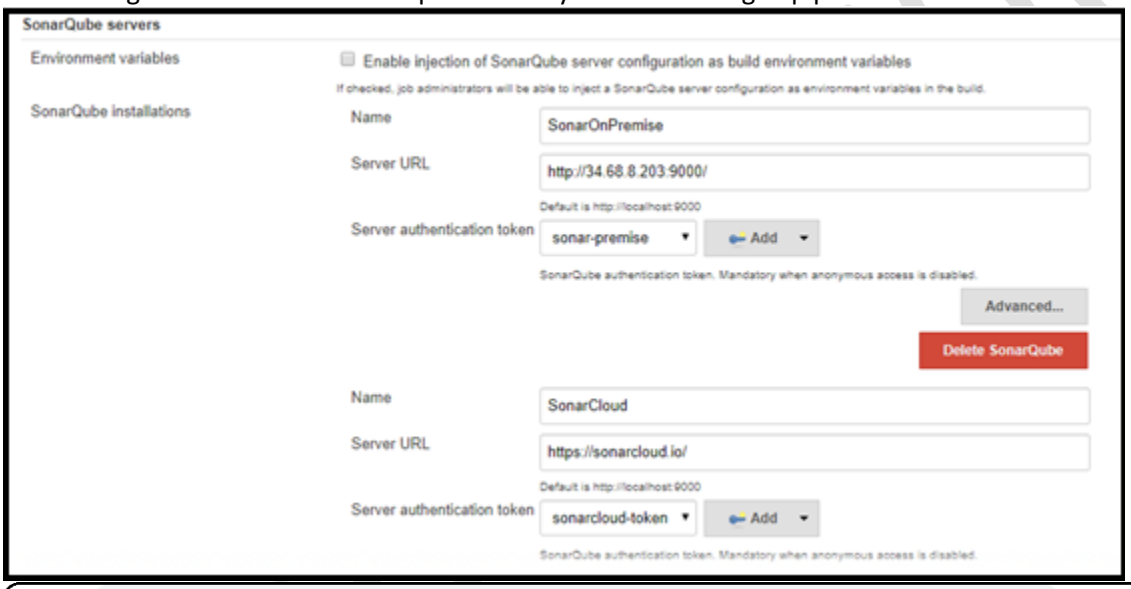
Install Plugin

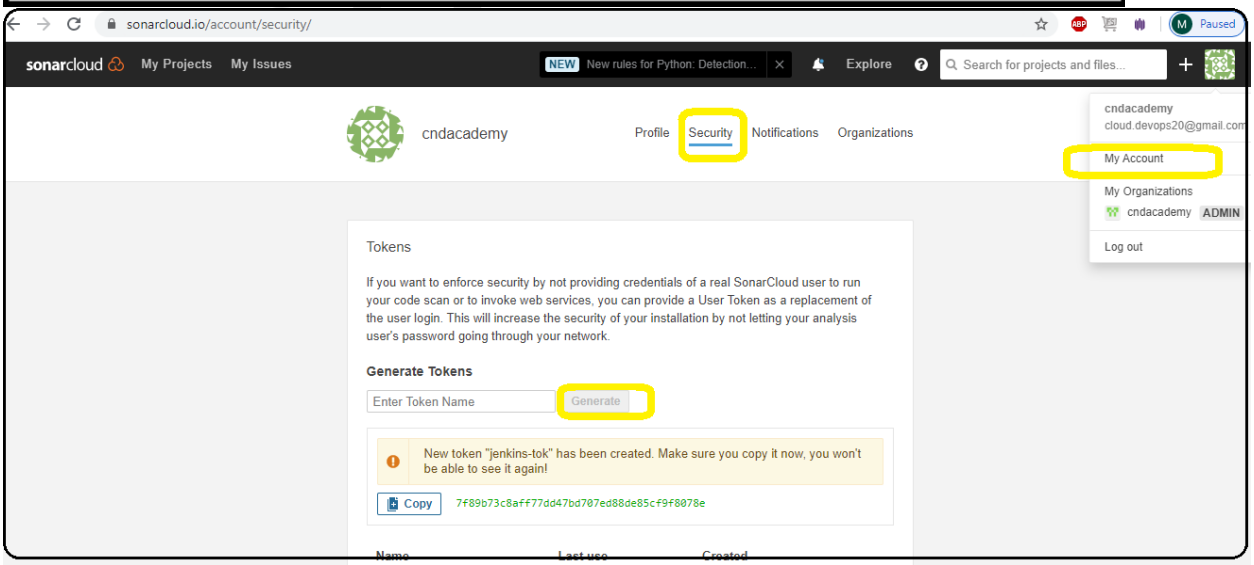
SonarQube Scanner for Jenkins

This plugin allows an easy integration of SonarQube, the open source platform for Continuous Inspection of code quality.

Configure Sonar Server: Manage Jenkins→ Configure System

Here configure what kind of sonarqube server you will be using in pipeline





Configure below project properties in pom.xml file

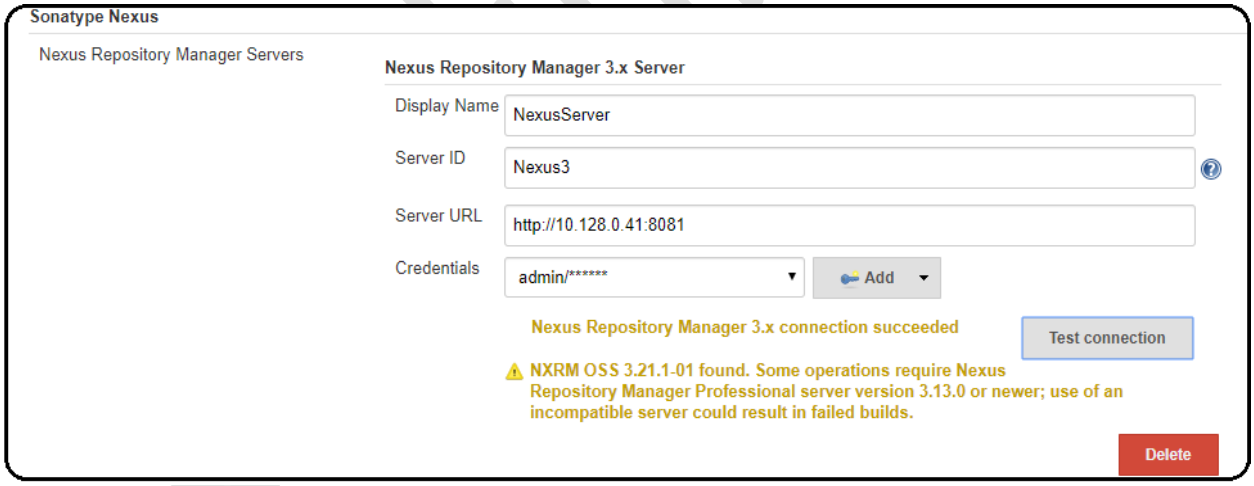
```
<properties>
  <sonar.projectKey>cndacademy_hello-world</sonar.projectKey>
  <sonar.organization>cndacademy</sonar.organization>
  <sonar.host.url>https://sonarcloud.io</sonar.host.url>
  <sonar.login>9090d9649f4fd0b138005d1cd39b9a7e89674536</sonar.login>
</properties>
```

Jenkins Interation with Nexus

Install Plugin

Nexus Platform Plugin

This plugin integrates Sonatype Nexus to Jenkins.



Jenkins › cicd-scripted-pipeline › Pipeline Syntax

- Back
- **Snippet Generator**
- Declarative Directive Generator
- Declarative Online Documentation
- Steps Reference
- Global Variables Reference
- Online Documentation
- Examples Reference
- IntelliJ IDEA GDSL

Overview

This **Snippet Generator** will help you learn the Pipeline Script code which can be used to define various steps. Pick a step you are interested in from the list, configure it, click **Generate Pipeline Script**, and you will see a Pipeline Script statement that would call the step with that configuration. You may copy and paste the whole statement into your script, or pick up just the options you care about. (Most parameters are optional and can be omitted in your script, leaving them at default values.)

Steps

Sample Step: nexusPublisher: Nexus Repository Manager Publisher

Nexus Repository Manager Publisher

Nexus Instance: NexusServer

Nexus Repository: maven-releases

Tag:

Packages

Group: com.efsavage

Artifact: hello-world-war

Version: 2.1

Packaging: war

Artifacts

Maven Artifact

File Path: target/hello-app.war

Classifier:

Extension:

Delete

Add Artifact Path

Delete

Add Package

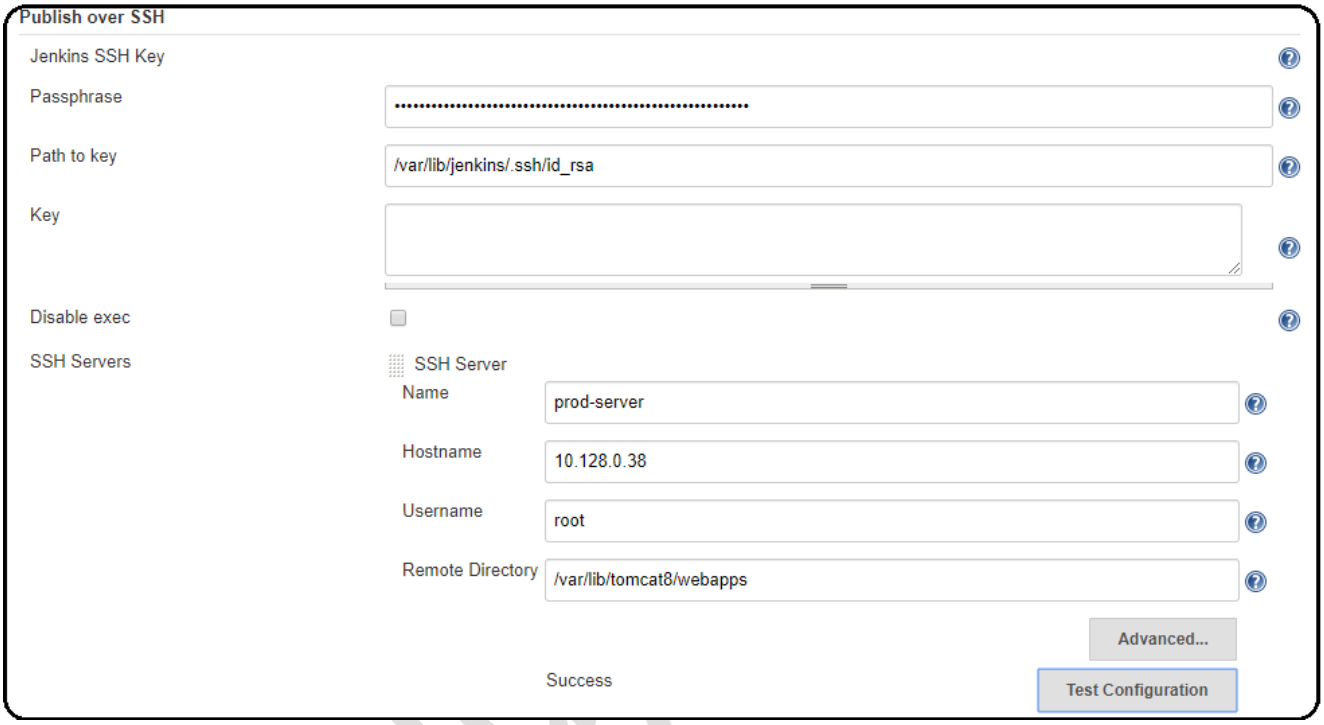
Generate Pipeline Script

```
nexusPublisher nexusInstanceId: 'Nexus3', nexusRepositoryId: 'maven-releases', packages: [[$class: 'MavenPackage', mavenAssetList: [[classifier: '', extension: '', filePath: 'target/hello-app.war']], mavenCoordinate: [artifactId: 'hello-world-war', groupId: 'com.efsavage', packaging: 'war', version: '2.1']]]
```

Jenkins Connection with Prod/Remote Server to copy file

Install Plugin

Publish Over SSH



Declarative Jenkinsfile script for complete pipeline

```
pipeline{
   agent any
   tools{
      maven 'maven3'
   }
   /*
   triggers{
      //cron('* * * * *')
   }
   */
   stages{
      stage("Checkout"){
         steps{
            git url: 'https://github.com/cloud-junction/devops-webapp-maven.git', branch: 'master',
credentialsId: 'github2'


            sh 'ls -ll'
         }
      }
      stage("Unit Test"){
         steps{
            sh 'mvn test'
         }
      }
      stage("Sonarqube Analysis"){
         steps{
```

Cloudjunction

```
            sh 'mvn verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectKey=devops-demo1'
        }
    }

    stage("Build"){
        steps{
            sh 'mvn package'
        }
    }
    stage("ArchiveArtifacts"){
        steps{
            archiveArtifacts artifacts: 'target/my-app.war', followSymlinks: false
        }
    }
    stage("PublishNexus"){
        steps{
            nexusPublisher nexusInstanceId: 'nexus3', nexusRepositoryId: 'maven-releases', packages: [[$class: 'MavenPackage', mavenAssetList: [[classifier: '', extension: '', filePath: 'target/my-app.war']], mavenCoordinate: [artifactId: 'my-app', groupId: 'com.mycompany.app', packaging: 'war', version: '2.0']]]
        }
    }
    stage("DeployDev"){
        steps{
            echo "Deploy Dev"
        }
    }
    stage("Approval"){
        steps{
```

Cloudjunction

```
            timeout(time: 15, unit: 'MINUTES'){
                input message: 'Do you approve deployment for production?' , ok: 'Yes'}


        }
    }
    stage("DeployProd"){
        steps{
            sshPublisher(publishers: [sshPublisherDesc(configName: 'dev-server', transfers:
[sshTransfer(cleanRemote: false, excludes: '', execCommand: '', execTimeout: 120000, flatten: false,
makeEmptyDirs: false, noDefaultExcludes: false, patternSeparator: '[, ]+', remoteDirectory: '',
remoteDirectorySDF: false, removePrefix: 'target', sourceFiles: 'target/my-app.war')],
usePromotionTimestamp: false, useWorkspaceInPromotion: false, verbose: false)])
        }
    }


  }
  post{
    failure {
        mail to:"cloud.junction.in.3@gmail.com",
        subject: "Status of pipeline: ${currentBuild.fullDisplayName}",
        body: "${env.BUILD_URL} has result FAILURE "
    }



  }
}
```

Cloudjunction

Dashboard › declarative-pipeline ›

Last Successful Artifacts

my-app.war 2.13 KB view

Recent Changes

Stage View

	Declarative: Checkout SCM	Declarative: Tool Install	Checkout	Unit Test	Sonarqube Analysis	Build	ArchiveArtifacts	PublishNexus	DeployDev	Approval	DeployProd
Average stage times:	624ms	56ms	769ms	4s	24s	5s	170ms	313ms	127ms	210ms	41ms
#10 Mar 05 14:45 No Changes	604ms	56ms	769ms	4s	23s	5s	138ms	306ms	136ms	216ms (paused for 14min 59s) aborted	38ms aborted
#9 Mar 05 14:39 1 commit	644ms	57ms	770ms	4s	26s	5s	203ms	320ms	118ms	205ms (paused for 14min 59s) aborted	44ms aborted

Permalinks

- Last build (#10), 25 min ago
- Last stable build (#8), 35 min ago
- Last successful build (#8), 35 min ago
- Last failed build (#6), 39 min ago
- Last unsuccessful build (#10), 25 min ago
- Last completed build (#10), 25 min ago

Cloudjunc

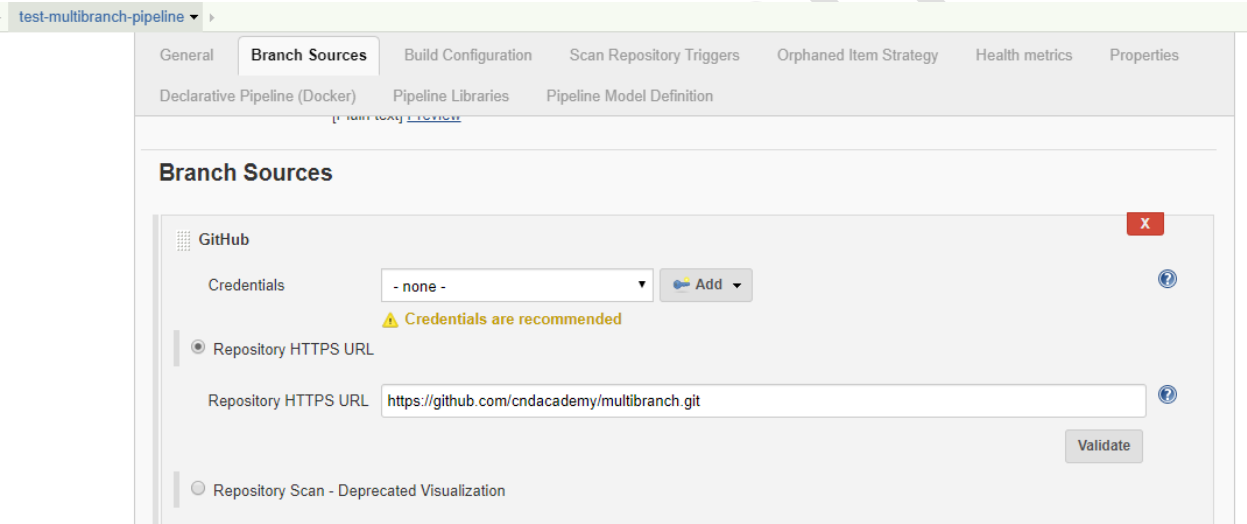
17. Multibranch pipeline:

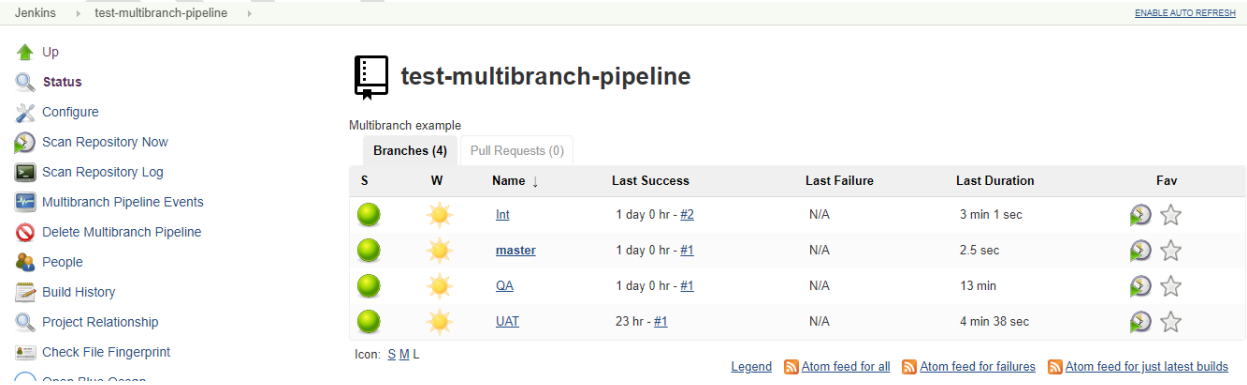
The purpose of the multibranch pipeline is to handle all branches in the repository. We make different pipeline for each branch and each branch should contains Jenkinsfile.

Suppose you want to perform complete CI/CD pipeline for the master branch and Only CI pipeline for the develop branch. You can do this with the help of multibranch pipeline project.

Create new project with multibranch pipeline type. Use sample repository

https://github.com/cndacademy/multibranch.git





18. Blue ocean

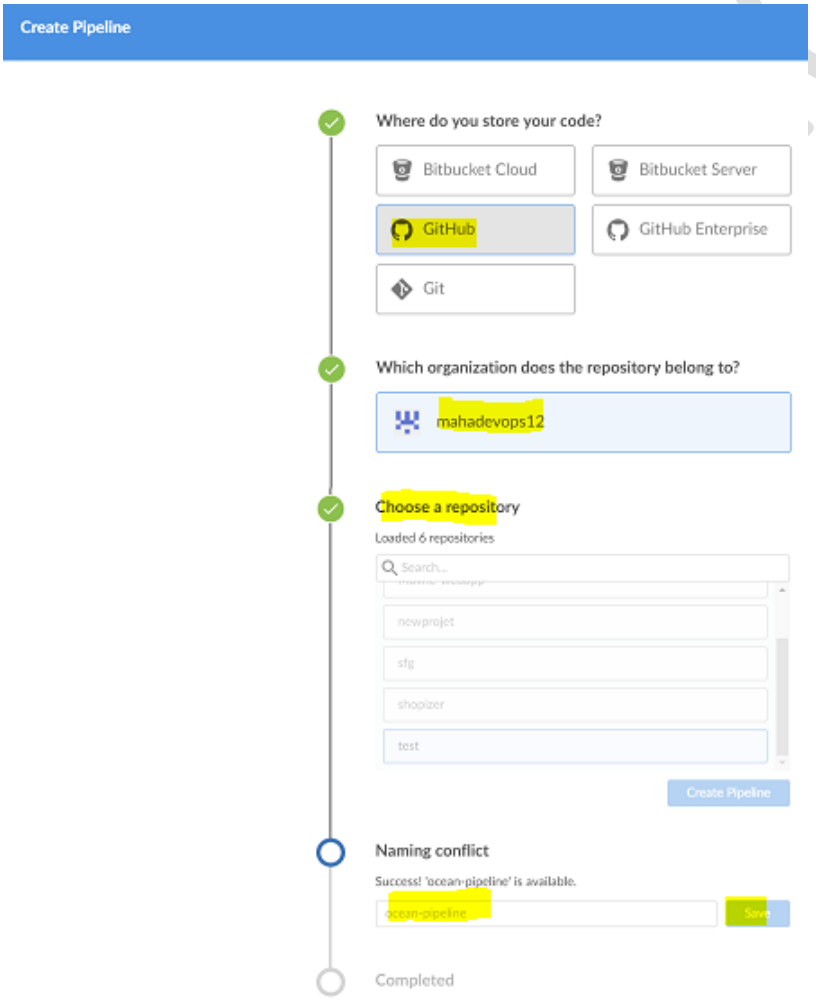
Blue Ocean's main features include:

- **Sophisticated visualizations** of continuous delivery (CD) Pipelines, allowing for fast and intuitive comprehension of your Pipeline's status.
- **Pipeline editor** - makes creation of Pipelines approachable by guiding the user through an intuitive and visual process to create a Pipeline.
- **Personalization** to suit the role-based needs of each member of the team.

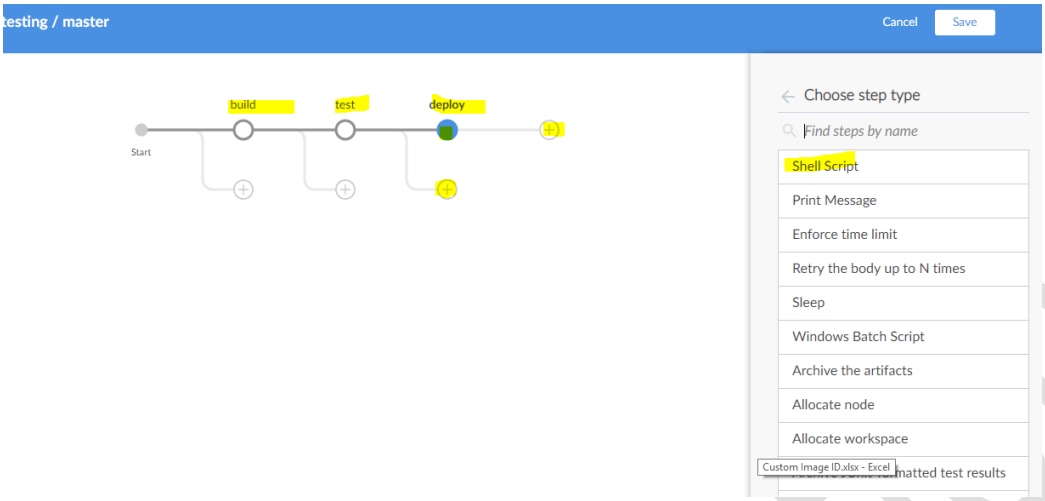
Install blueocean plugin

Refer the docs: https://jenkins.io/doc/book/blueocean/creating-pipelines/

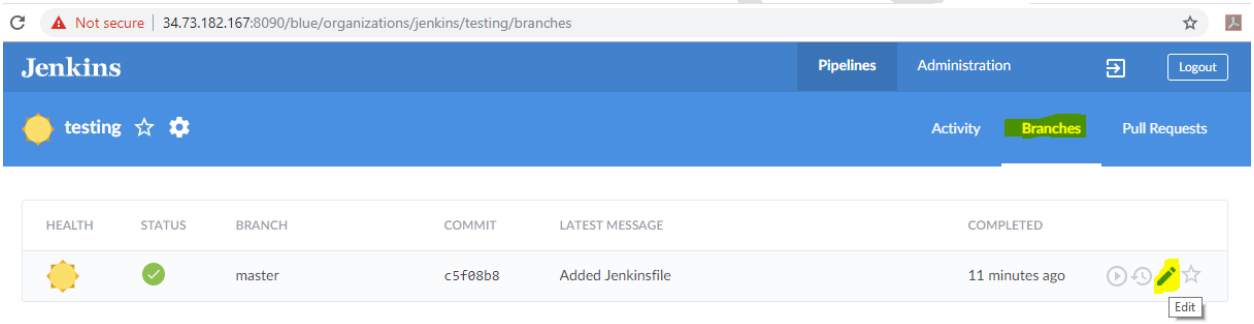
Click on Blue ocean from Jenkins Home



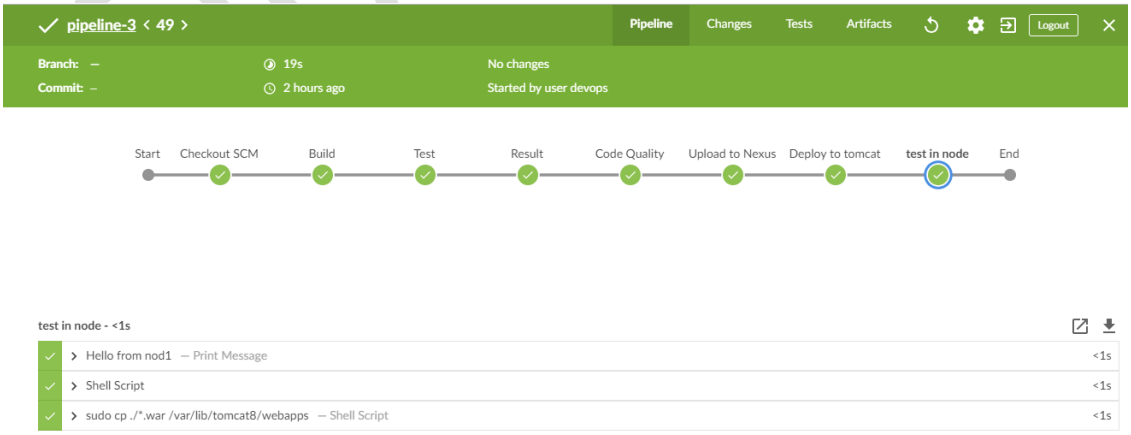
Create Job



Edit existing job



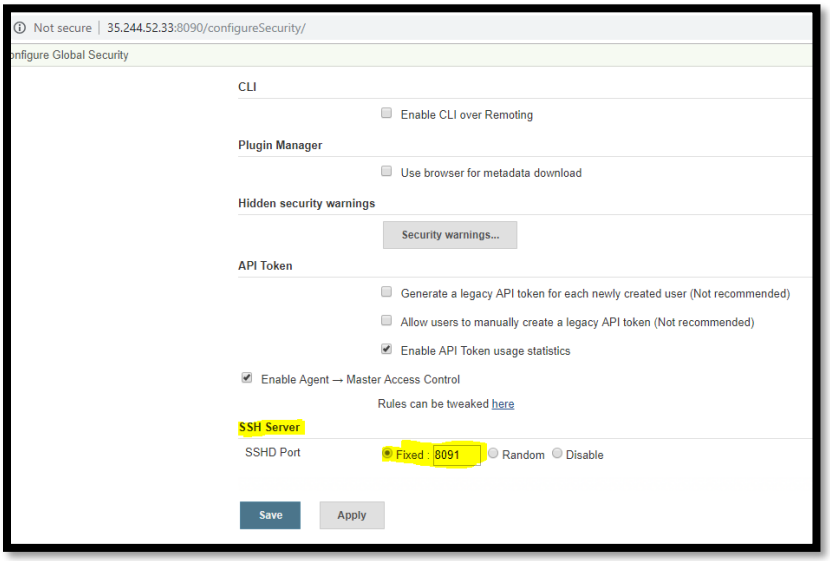
Final view



19. Jenkins CLI

Using the CLI over SSH

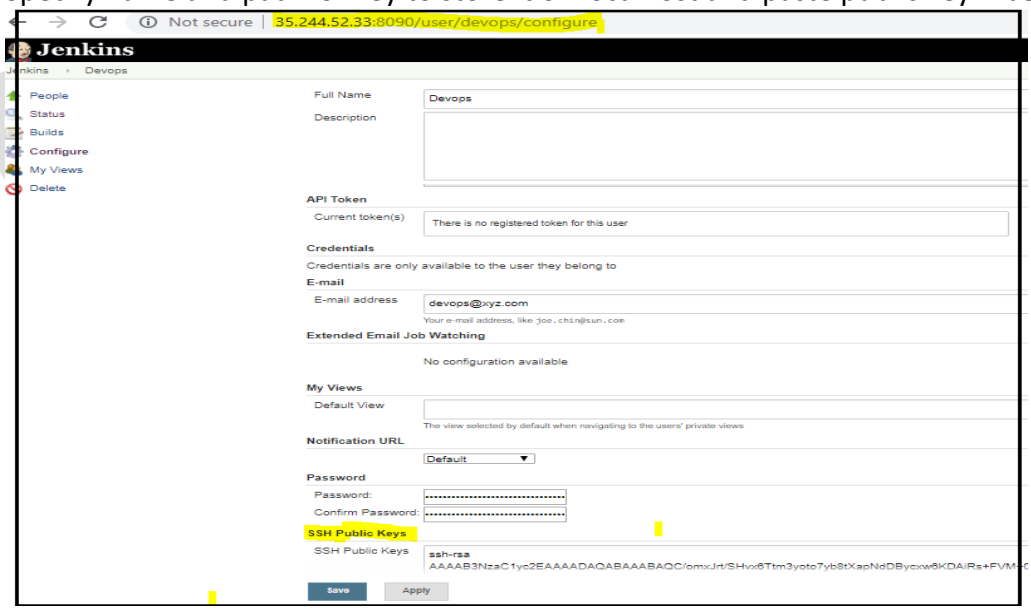
In a new Jenkins installation, the SSH service is disabled by default. You may choose to set a specific port or ask Jenkins to pick a random port in the **Configure Global Security page**. Also gives ssh port



1. Create user in jenkins with name devops

2. First create ssh keys for user
   # ssh-keygen -C devops (Here we are creating keys for user devops)

3. Specify name and path for key to store it on localhost and paste public key in Jenkins.



CLI commands

#ssh -l devops -i /root/.ssh/devops_private -p 8091 localhost help

Here you will get the all available options

List jobs

ssh -l devops -i /root/.ssh/devops_private -p 8091 localhost list-jobs

Build Job

ssh -l devops -i /root/.ssh/devops_private -p 8091 localhost build Helloworld

Show build job console output

ssh -l devops -i /root/.ssh/devops_private -p 8091 localhost console Helloworld

Cloudjunction

20. Extra Preparation for Jenkinsfile (If you want more dig into groovy)

String interpolation

Jenkins Pipeline uses rules identical to Groovy for string interpolation. Groovy's String interpolation support can be confusing to many newcomers to the language. While Groovy supports declaring a string with either single quotes, or double quotes, for example:

```
def username = "Cloudjunction"

def company = "XYZ"
```

How to access variable in pipeline

```
echo "Hello Mr. ${username} , Welcome to ${company} !!"
```

Example

```
def username = "Ajit"

def company = "XYZ"

pipeline {

   agent any

   stages {

        stage('Test') {

        steps {

            echo "Hello Mr. ${username} , Welcome to ${company} !!"

            echo 'Testing..'

        }

      }

     }

}
```

Using environment variables

Cloudjunction

Jenkins Pipeline exposes environment variables via the global variable env, which is available from anywhere within a Jenkinsfile. The full list of environment variables accessible from within Jenkins Pipeline is documented at

http://localhost:8080/pipeline-syntax/globals#env

In linux you can find all environment by

$env or $printenv

Environment variables are accessible from Groovy code as env.VARNAME or simply as VARNAME

Example1

```
node {
   stage('test'){
  echo 'Hello World'
  echo "${ BUILD_URL}"
   }
}
```

Example2

Cloudjunction

```
def username = "Cloudjunction"
def company = "XYZ"
pipeline {
   agent any
   stages {


      stage('Example') {
         steps {
            echo "Hello Mr. ${username} , Welcome to ${company} !!"
            echo "Running ${env.BUILD_ID} on ${env.JENKINS_URL}"
            echo 'Testing..'
            mail to: 'abc@gmail.com',
                  subject: "Job '${JOB_NAME}' (${BUILD_NUMBER}) is waiting for input",
                  body: "Please go to ${BUILD_URL} and verify the build"
         }
      }


   }
}
```

Cloudjunction

Setting environment variables

Setting an environment variable within a Jenkins Pipeline is accomplished differently depending on whether Declarative or Scripted Pipeline is used.

Declarative Pipeline supports an environment directive, whereas users of Scripted Pipeline must use the withEnv step.

Example

```
pipeline {
   agent any
   environment {
      myenv= 'dev'
   }
   stages {
      stage('Example') {

         steps {
            sh 'printenv'
         }
      }
   }
}
```

Cloudjunction

Now using **Scripted Pipeline**

```
node {

  withEnv(["PATH+MAVEN=${tool 'M3'}/bin"]) {
    sh 'mvn -B verify'
  }
}
```

Handling credentials

For secret text, usernames and passwords, and secret files

First Create credentials in Jenkins credentials as per below

Example

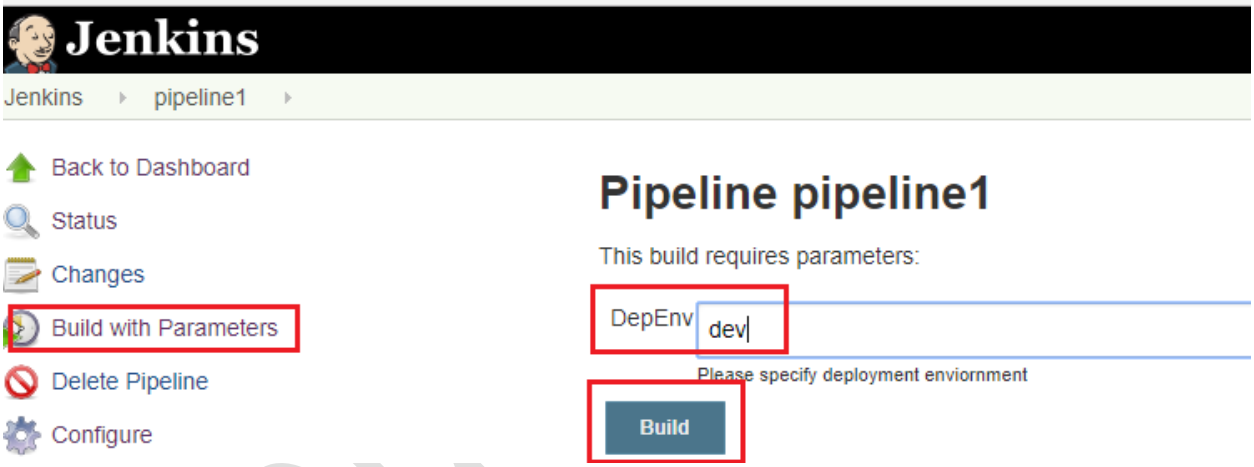
```
pipeline {
   agent any
   environment {
      AWS_ACCESS_KEY_ID     = credentials('jenkins-aws-secret-key-id')
      AWS_SECRET_ACCESS_KEY = credentials('jenkins-aws-secret-access-key')
   }
   stages {
      stage('Example stage 1') {
         steps {
            //
            echo "Hello"
         }
      }

   }
}
```

Cloudjunction

Handling parameters

Declarative Pipeline supports parameters out-of-the-box, allowing the Pipeline to accept user-specified parameters at runtime via the parameters directive. Configuring parameters with Scripted Pipeline is done with the properties step, which can be found in the Snippet Generator.

If you configured your pipeline to accept parameters using the **Build with Parameters** option, those parameters are accessible as members of the params variable.



Example

```
pipeline {
   agent any
   parameters {
      string(name: 'DepEnv', defaultValue: 'dev', description: 'Please specify deployment
enviornment')
   }
   stages {
      stage('Example') {
         steps {
            echo " Deploying applciation in ${params.DepEnv} enviornment !! !"
         }
      }
   }
}
```

Cloudjunction

Handling failure

Declarative Pipeline supports robust failure handling by default via its post section which allows declaring a number of different "post conditions" such as: always, unstable, success, failure, and changed

Example

```
pipeline {
   agent any
   stages {
      stage('Example') {
         steps {
            echo 'Hello World'

            sh 'fdrhhsh'
         }
      }
   }
   post {


      always {
         echo "I am in always post"
         echo 'I will always say Hello again!'
      }

      failure{
         echo "I am in failure post"
         mail to: 'mahadeva.garad1@gmail.com',
                  subject: " Pipeline failed for Job '${JOB_NAME}' (${BUILD_NUMBER}) ",
                           body: "Please go to ${BUILD_URL} and verify the build"
      }

      changed{

         echo "I am in changed post actions"
      }

    }
}
```

Cloudjunction

Conditions meaning

always

Run the steps in the post section regardless of the completion status of the Pipeline's or stage's run.

changed

Only run the steps in post if the current Pipeline's or stage's run has a different

aborted

Only run the steps in post if the current Pipeline's or stage's run has an "aborted" status, usually due to the Pipeline being manually aborted. This is typically denoted by gray in the web UI.

failure

Only run the steps in post if the current Pipeline's or stage's run has a "failed" status, typically denoted by red in the web UI.

success

Only run the steps in post if the current Pipeline's or stage's run has a "success" status, typically denoted by blue or green in the web UI.

Scripted Pipeline however relies on Groovy's built-in try/catch/finally semantics for handling failures during execution of the Pipeline.

Example

```
node {

   stage('Test') {
      try {
         echo "Im in try block"
         sh 'szxderrgertgb'


      }
      finally {
        echo "Im in finally bloack"
      }
   }

}
```

Cloudjunction

Running Pipeline on Specific slave

Example

```
pipeline {
   agent {
            label 'test'
         }
   stages {
      stage('Example') {
         steps {
            echo 'Hello World'

            sh 'mvn -version'
         }
      }
   }

}
```

Cloudjunction

Jenkins pipeline syntax

options

The options directive allows configuring Pipeline-specific options from within the Pipeline itself. Pipeline provides a number of these options, such as buildDiscarder, but they may also be provided by plugins, such as timestamps.

```
pipeline{
   agent any

   options{
      timeout(time: 1, unit: 'HOURS')//Specifying a global execution timeout of one hour, after which Jenkins will abort the Pipeline run.

      //timestamps()   //Prepend all console output generated by the Pipeline run with the time at which the line was emitted
      //retry(3)     On failure, retry the entire Pipeline the specified number of times
   }
   stages{

      stage('options_test'){

         steps{

          echo "Hello from options"
         // sh "sfcee"
         }
      }
   }

}
```

Cloudjunction

parameters

The parameters directive provides a list of parameters which a user should provide when triggering the Pipeline. The values for these user-specified parameters are made available to Pipeline steps via the params object

```
pipeline {

   agent any

   parameters{

      string(name: 'PERSON', defaultValue: 'Mahadev', description: 'Whom should I say Hello')
      text(name: 'BIOGRAPHY', defaultValue: 'B +ve', description: 'Enter some biographical
information')
      booleanParam(name: 'TOGGLE', defaultValue: true, description: 'Toggle this value')
      choice(name: 'CHOICE', choices: ['One', 'Two', 'Three'], description: 'Pick something')
      password(name: 'PASSWORD', defaultValue: 'SECRET', description: 'Enter a password')
      file(name: "FILE", description: "Choose a file to upload")
   }
   stages{
      stage('parameter_test'){

         steps{
            echo "Hello ${params.PERSON}"
            echo "Biography: ${params.BIOGRAPHY}"
            echo "Toggle: ${params.TOGGLE}"
             echo "Choice: ${params.CHOICE}"
              echo "Password: ${params.PASSWORD}"

         }
      }
   }
}
```

Cloudjunction

triggers

The triggers directive defines the automated ways in which the Pipeline should be re-triggered. For Pipelines which are integrated with a source such as GitHub or BitBucket, triggers may not be necessary as webhooks-based integration will likely already be present. The triggers currently available are cron, pollSCM and upstream.

cron

Accepts a cron-style string to define a regular interval at which the Pipeline should be re-triggered, for example: triggers { cron('H */4 * * 1-5') }

pollSCM

Accepts a cron-style string to define a regular interval at which Jenkins should check for new source changes. If new changes exist, the Pipeline will be re-triggered. For example: triggers { pollSCM('H */4 * * 1-5') }

Example

Jenkins cron syntax

The Jenkins cron syntax follows the syntax of the cron utility (with minor differences). Specifically, each line consists of 5 fields separated by TAB or whitespace:

MINUTE	HOUR	DOM	MONTH	DOW
Minutes within the hour (0–59)	The hour of the day (0–23)	The day of the month (1–31)</td>	The month (1–12)	The day of the week (0–7) where 0 and 7 are Sunday.</td>

To specify multiple values for one field, the following operators are available. In the order of precedence,

- * specifies all valid values
- M-N specifies a range of values
- M-N/X or */X steps by intervals of X through the specified range or whole valid range
- A,B,…,Z enumerates multiple values

To allow periodically scheduled tasks to produce even load on the system, the symbol H (for “hash”) should be used wherever possible. For example, using 0 0 * * * for a dozen daily jobs will cause a large spike at midnight. In contrast, using H H * * *would still execute each job once a day, but not all at the same time, better using limited resources.

The H symbol can be used with a range. For example, H H(0-7) * * * means some time between 12:00 AM (midnight) to 7:59 AM. You can also use step intervals with H, with or without ranges.

The H symbol can be thought of as a random value over a range, but it actually is a hash of the job name, not a random function, so that the value remains stable for any given project

In addition, @yearly, @annually, @monthly, @weekly, @daily, @midnight, and @hourly are supported as convenient aliases. These use the hash system for automatic balancing. For example, @ @midnight actually means some time between 12:00 AM and 2:59 AM.

Cron Scheduling examples

triggers{ cron('H/15 * * * *') }

every fifteen minutes (perhaps at :07, :22, :37, :52)

triggers{ H(0-29)/10 * * * *) }

every ten minutes in the first half of every hour (three times, perhaps at :04, :14, :24)

triggers{ 45 9-16/2 * * 1-5) }

once every two hours at 45 minutes past the hour starting at 9:45 AM and finishing at 3:45 PM every weekday.

triggers{ H H(9-16)/2 * * 1-5) }

once in every two hours slot between 9 AM and 5 PM every weekday (perhaps at 10:38 AM, 12:38 PM, 2:38 PM, 4:38 PM)

triggers{ H H 1,15 1-11 *) }

once a day on the 1st and 15th of every month except December

cloudjunction

Example

```
pipeline {
   agent any
   triggers {
      cron('* * * * *')
   }
   stages {
      stage('Example') {
         steps {
            echo 'Hello World'
         }
      }
   }
}
```

Cloudjunction

tools

A section defining tools to auto-install and put on the PATH. This is ignored if agent none is specified.

Example

```
pipeline {
   agent any
   tools {
      maven 'M3'
   }
   stages {
      stage('Example') {
         steps {
            sh 'mvn --version'
         }
      }
   }
}
```

The tool name must be pre-configured in Jenkins under **Manage Jenkins** → **Global Tool Configuration**

Cloudjunction

input

The input directive on a stage allows you to prompt for input, using the input step. The stage will pause after any optionshave been applied, and before entering the stage`s `agent or evaluating its when condition. If the input is approved, the stage will then continue. Any parameters provided as part of the input submission will be available in the environment for the rest of the stage

Example

```
pipeline {
   agent any
   stages {
      stage('Example') {
         input {
            message "Should we continue?"
            //ok "Yes, we should."
            //submitter "mahadev"
           // parameters {
          //     string(name: 'PERSON', defaultValue: 'Mr Jenkins', description: 'Who should I say
hello to?')
            //}
         }
         steps {
            echo "Hello, nice to meet you."
         }
      }
   }
}
```

Cloudjunction

Console Output

```
Started by user Mahadev Garad
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/Testpipeline
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Example)
[Pipeline] input
Should we continue?
Proceed or Abort
```

Should we continue?

PERSON Mr Jenkins

Who should I say hello to?

Yes, we should. Abort

when

The when directive allows the Pipeline to determine whether the stage should be executed depending on the given condition. The when directive must contain at least one condition. If the when directive contains more than one condition, all the child conditions must return true for the stage to execute

```
pipeline {
   agent any
   stages {
      stage('Example Build') {
         steps {
            echo 'Hello World'
         }
      }
      stage('Example Deploy') {
         when {
            branch 'production'
            environment name: 'DEPLOY_TO', value: 'production'
           // anyOf {
            //    environment name: 'DEPLOY_TO', value: 'production'
               environment name: 'DEPLOY_TO', value: 'staging'
           // }
         }
         steps {
            echo 'Deploying'
         }
      }
   }
}
```

Evaluating when before entering the stage's `agent

By default, the when condition for a stage will be evaluated after entering the agent for that stage, if one is defined

Cloudjunction

Parallel

Stages in Declarative Pipeline may declare a number of nested stages within a parallel block, which will be executed in parallel. Note that a stage must have one and only one of steps, stages, or parallel. The nested stages cannot contain further parallel stages themselves, but otherwise behave the same as any other stage, including a list of sequential stages within stages. Any stage containing parallel cannot contain agent or tools, since those are not relevant without steps.

In addition, you can force your parallel stages to all be aborted when one of them fails, by adding failFast true to the stagecontaining the parallel

```
pipeline {
   agent any
   stages {
      stage('Non-Parallel Stage') {
         steps {
            echo 'This stage will be executed first.'
         }
      }
      stage('Parallel Stage') {
        // when {
        //     branch 'master'
        // }
        // failFast true
         parallel {
            stage('Branch A') {
              // agent {
               //    label "for-branch-a"
              // }
               steps {
                  echo "On Branch A"
               }
            }
            stage('Branch B') {
              // agent {
              //     label "for-branch-b"
             //   }
               steps {
                  echo "On Branch B"
               }
```

cloudjunction

```
            }
            stage('Branch C') {
              // agent {
               //    label "for-branch-c"
              // }
               stages {
                  stage('Nested 1') {
                     steps {
                        echo "In stage Nested 1 within Branch C"
                     }
                  }
                  stage('Nested 2') {
                     steps {
                        echo "In stage Nested 2 within Branch C"
                     }
                  }
               }
            }
         }
      }
   }
}
```

Cloudjunction

Flow Control

Scripted Pipeline is serially executed from the top of a Jenkinsfile downwards, like most traditional scripts in Groovy or other languages. Providing flow control therefore rests on Groovy expressions, such as the if/else conditionals, for example:

Example

```
node {
  stage('Example') {
    if (env.BRANCH_NAME == 'master') {
      echo 'I only execute on the master branch'
    } else {
      echo 'I execute elsewhere'
    }
  }
```

Cloudjunction