## **What is Jenkinsfile?**

****Jenkinsfile**** is just a text file, usually checked in along with the project’s source code in Git repo. Ideally, every application will have its own Jenkinsfile.

Jenkinsfile can be written in two aspects – ****Scripted pipeline syntax & Declarative pipeline syntax****

**What is Jenkins Scripted Pipeline?**

Jenkins pipelines are traditionally written as scripted pipelines. Ideally, the scripted pipeline is stored in Jenkins webUI as a Jenkins file. The end-to-end scripted pipeline script is written in Groovy.

1. It requires knowledge of Groovy programming as a prerequisite.
2. Jenkinsfile starts with the word ****node****.
3. Can contain standard programming constructs like if-else block, try-catch block, etc.

****Sample Scripted Pipeline****

|  |  |
| --- | --- |
| 1  2  3  4  5 | node {      stage('Stage 1') {          echo 'hello'      }  } |

## **What is Jenkins Declarative Pipeline?**

The Declarative Pipeline subsystem in Jenkins Pipeline is relatively new, and provides a simplified, opinionated syntax on top of the Pipeline subsystems.

* The latest addition in Jenkins pipeline job creation technique.
* Jenkins declarative pipeline needs to use the predefined constructs to create pipelines. Hence, it is not flexible as a scripted pipeline.
* Jenkinsfile starts with the word ****pipeline****.

Jenkins declarative pipeline should be the preferred way to create a Jenkins job as they offer a rich set of features, come with less learning curve & no prerequisite to learn a programming language like Groovy just for the sake of writing pipeline code.

We can also validate the syntax of the Declarative pipeline code before running the job. It helps to avoid a lot of runtime issues with the build script.

### Our First Declarative Pipeline

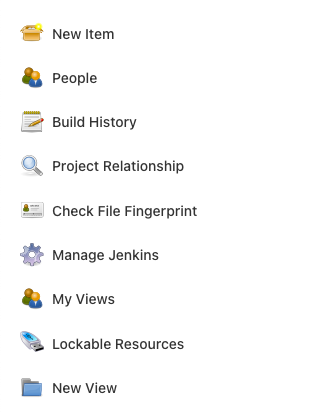
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | pipeline {      agent any      stages {          stage('Welcome Step') {              steps {                  echo 'Welcome to LambdaTest'              }          }      }  } |

Few recommend VS Code IDE for writing Jenkins pipeline scripts, especially when creating Jenkins Declarative pipeline examples.

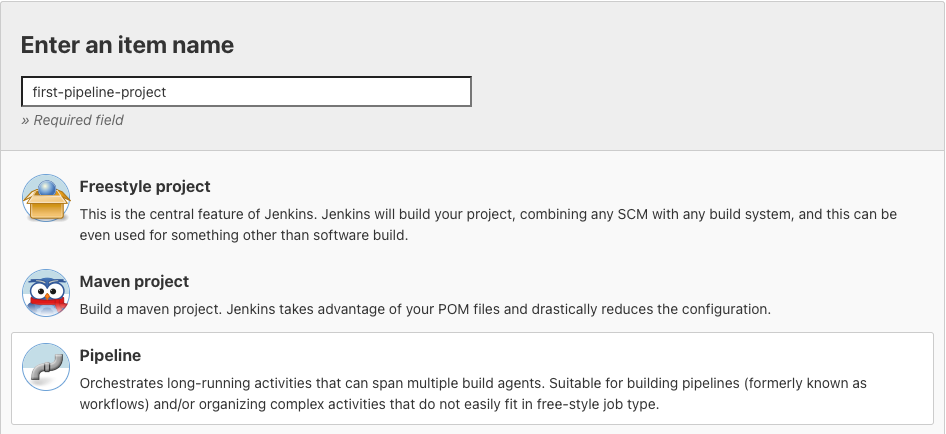
## **Running Your First Declarative Pipeline**

In this section, we will learn how to run a Jenkins declarative pipeline. Let us run our Jenkins declarative pipeline step by step.

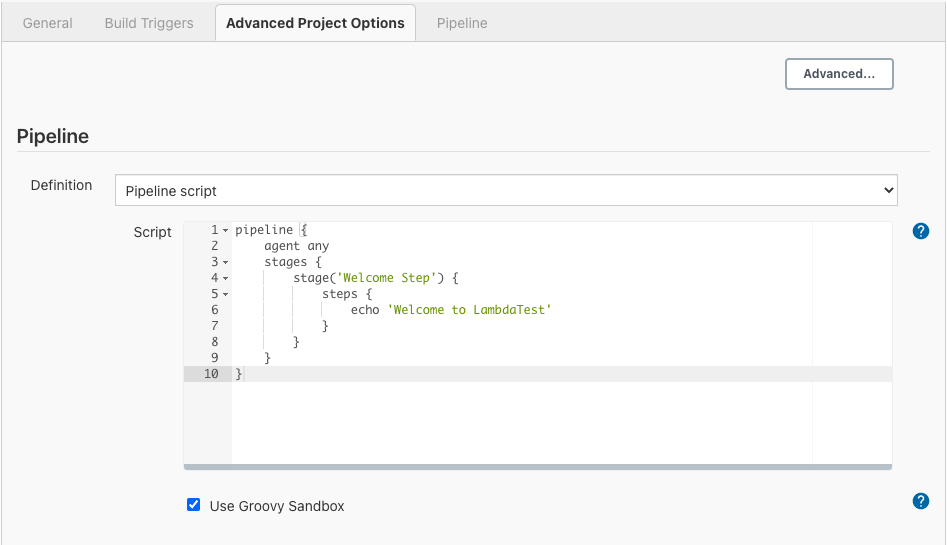
****Step 1:**** Open Jenkins home page (http://localhost:8080 in local) & click on ****New Item**** from the left side menu.



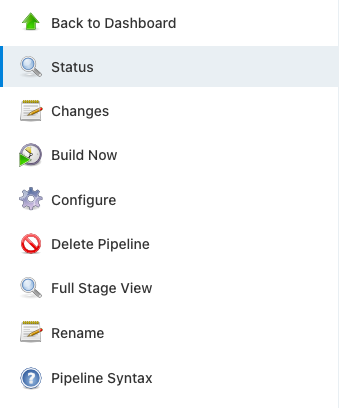
****Step 2:**** Enter ****Jenkins job name**** & choose the style as ****Pipeline**** & click ****OK****.



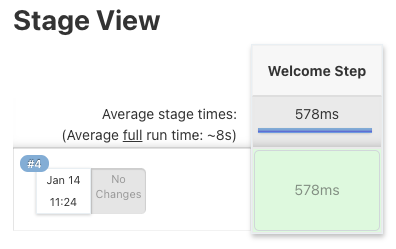
****Step 3:**** Scroll down to the ****Pipeline**** section & copy-paste your first Declarative style Pipeline code from below to the script textbox.



****Step 4:**** Click on the Save button & Click on ****Build Now**** from the left side menu.

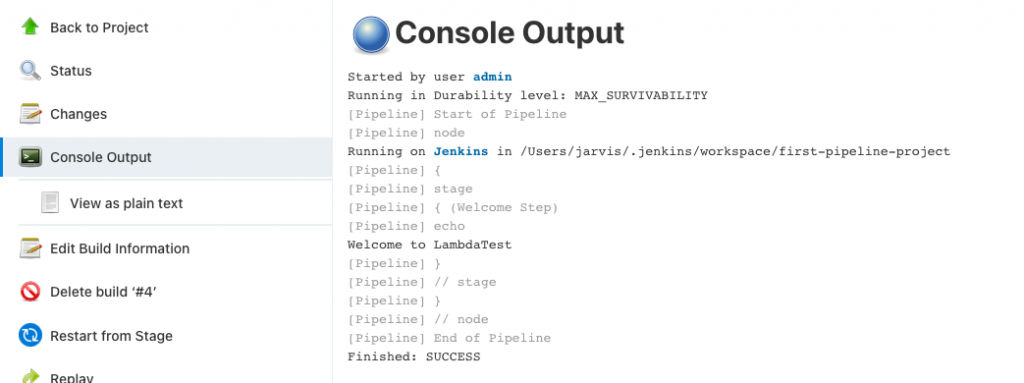


We can see the build running stage by stage in ****Stage**** View.



****Step 5:**** To check logs from Build Job, click on any stage & click on the ****check logs**** button. Or you can use the ****Console Output**** from the left side menu to see the logs for the build.

### Console Output



## **Jenkins Declarative Pipeline Syntax**

In this section, we will look at the most commonly used Jenkins declarative pipeline examples or syntax. Typically, declarative pipelines contain one or more declarative steps or directives, as explained below.

****pipeline****

Entire Declarative pipeline script should be written inside the ****pipeline**** block. It’s a ****mandatory**** block.

|  |  |
| --- | --- |
| 1  2 | pipeline {  } |

****agent****  
Specify where the Jenkins build job should run. agent can be at pipeline level or stage level. It’s mandatory to define an agent.

****Possible values-****

1. ****any**** – Run Job or Stage on any available agent.

|  |  |
| --- | --- |
| 1  2  3 | pipeline {       agent any  } |

1. ****none**** – Don’t allocate any agent globally for the pipeline. Every stage should specify their own agent to run.

|  |  |
| --- | --- |
| 1  2  3 | pipeline {       agent none  } |

1. ****label**** – Run the job in agent which matches the label given here. Remember [Jenkins CI/CD](https://www.lambdatest.com/blog/what-is-jenkins/" \t "https://www.lambdatest.com/blog/jenkins-declarative-pipeline-examples/_blank) can work on Master/Agent architecture. Master nodes can delegate the jobs to run in Agent nodes. Nodes on creation given a name & label to identify them later.

E.g All linux nodes can be labeled as linux-machine

|  |  |
| --- | --- |
| 1  2  3  4  5 | pipeline {         agent {             label 'linux-machine'         }  } |

1. ****docker**** – Run the job in given Docker container
2. Stages

****stages**** block constitutes different executable stage blocks. At least one stage block is mandatory inside stages block.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | pipeline {         agent {             label 'linux-machine'         }       stages {       }  } |

## **stage**

stage block contains the actual execution steps. Stage block has to be defined within stages block. It’s mandatory to have at least one stage block inside the stage block. Also its mandatory to name each stage block & this name will be shown in the ****Stage View**** after we run the job.

In below example, the stage is named as “****build step****”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | pipeline {         agent {             label 'linux-machine'         }       stages {           stage('build step') {           }       }  } |

## **steps**

****steps**** block contains the actual build step. It’s mandatory to have at least one step block inside a stage block.

Depending on the Agent’s operating system (where Jenkins job runs), we can use shell, bat, etc., inside the steps command.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | pipeline {       agent any       stages {           stage('build step') {                steps {                   echo "Build stage is running"                }           }       }  } |

Scripted Pipeline in Jenkins job

## **parameters**

The parameters directive provides a way for Jenkins job to interact with Jenkins CI/CD users during the running of the build job.

parameter can be of the following types – ****string, text, booleanParam, choice**** & ****password****

****string****– Accepts a value of String type from Jenkins user.

E.g.  
****string(name: ‘NAME’, description: ‘Please tell me your name?’)****

****text**** – Accepts multi line value from Jenkins user  
E.g.  
****text(name: ‘DESC’, description: ‘Describe about the job details’)****

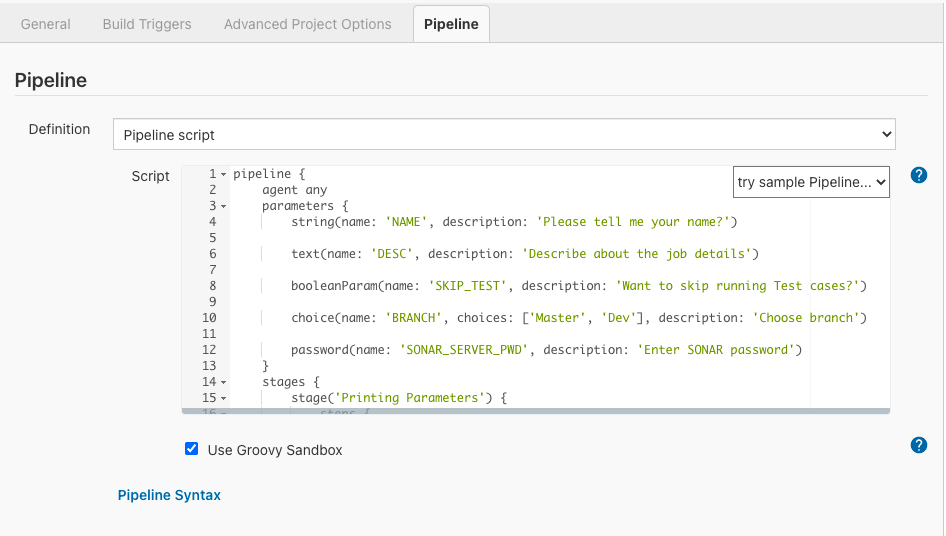
****booleanParam**** – Accepts a true/false value from Jenkins user  
E.g.  
****booleanParam(name: ‘SKIP\_TEST’, description: ‘Want to skip running Test cases?’)****

****choice****– Jenkins user can choose one among the choices of value provided  
E.g.  
****choice(name: ‘BRANCH’, choices: [‘Master’, ‘Dev’], description: ‘Choose the branch’)****

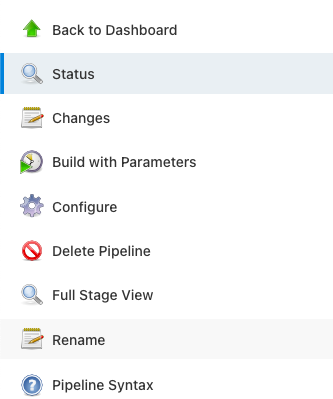
****password****– Accepts a secret like password from Jenkins user  
E.g.  
****password(name: ‘SONAR\_SERVER\_PWD’, description: ‘Enter SONAR password’)****

Let’s look at a sample on how to use parameters directive-

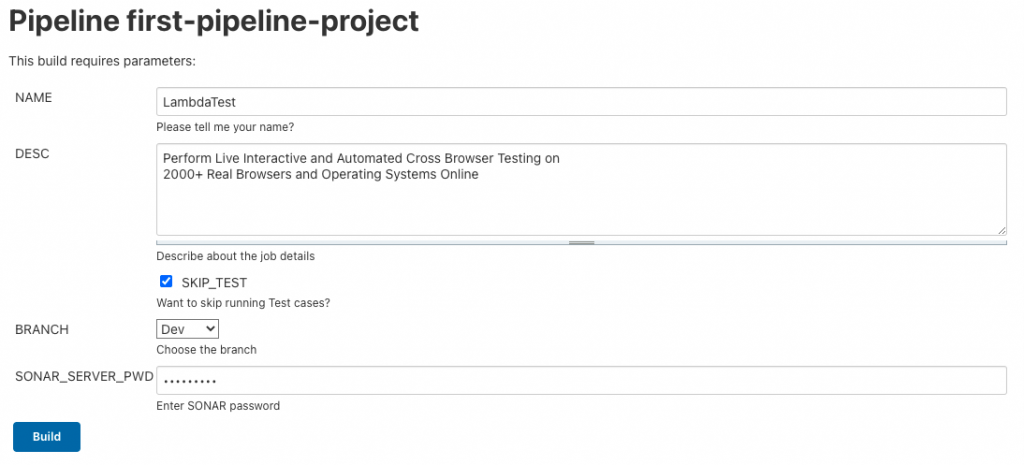
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | pipeline {      agent any      parameters {          string(name: 'NAME', description: 'Please tell me your name?')            text(name: 'DESC', description: 'Describe about the job details')            booleanParam(name: 'SKIP\_TEST', description: 'Want to skip running Test cases?')            choice(name: 'BRANCH', choices: ['Master', 'Dev'], description: 'Choose branch')            password(name: 'SONAR\_SERVER\_PWD', description: 'Enter SONAR password')      }      stages {          stage('Printing Parameters') {              steps {                  echo "Hello ${params.NAME}"                    echo "Job Details: ${params.DESC}"                    echo "Skip Running Test case ?: ${params.SKIP\_TEST}"                    echo "Branch Choice: ${params.BRANCH}"                    echo "SONAR Password: ${params.SONAR\_SERVER\_PWD}"              }          }      }  } |



If a parameters directive is used in the pipeline, Jenkins CI/CD can sense that it needs to accept the user’s input while running the job. Hence Jenkins will change the ****Build now**** link in the left side menu to ****Build with parameters**** link.



When we click on the ****Build with Parameters**** link, Jenkins CI/CD will let us pass values for the parameters we configured in the declarative pipeline.

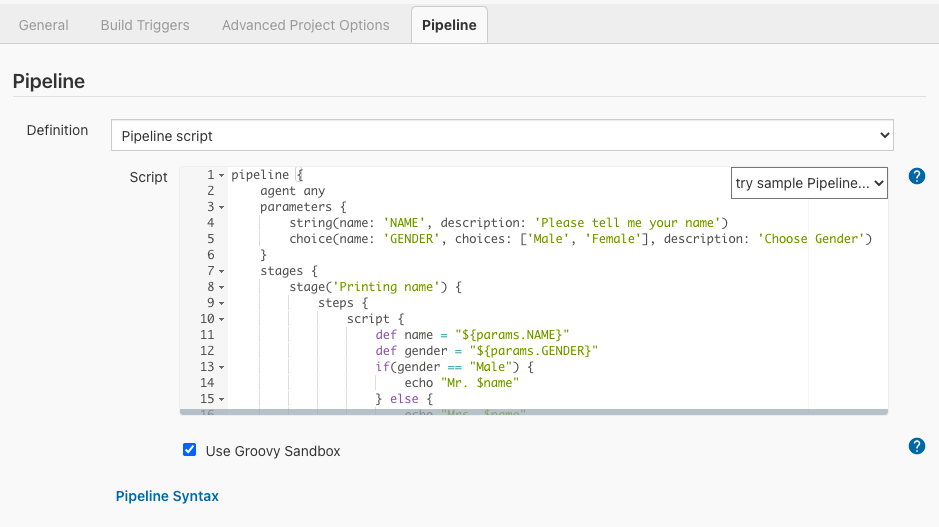


Once we enter each parameter’s values, we can hit the ****build**** button to run the job.

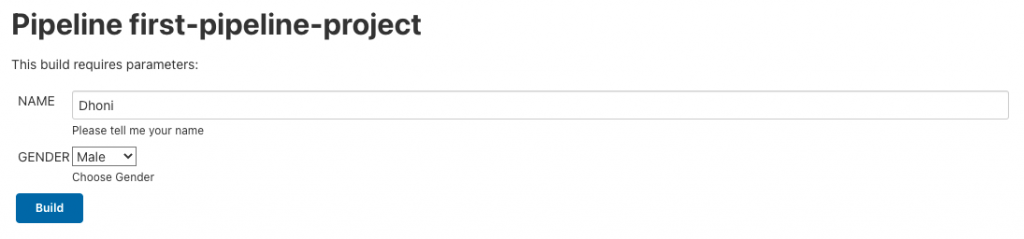
## **script**

script block helps us run ****Groovy**** code inside the Jenkins declarative pipeline.

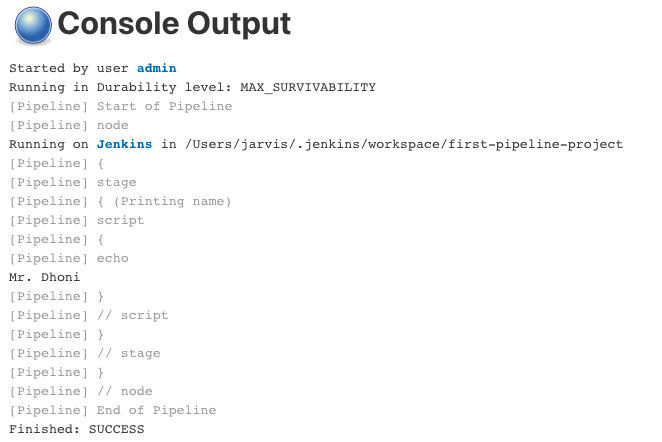
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | pipeline {      agent any      parameters {          string(name: 'NAME', description: 'Please tell me your name')          choice(name: 'GENDER', choices: ['Male', 'Female'], description: 'Choose Gender')      }      stages {          stage('Printing name') {              steps {                  script {                      def name = "${params.NAME}"                      def gender = "${params.GENDER}"                      if(gender == "Male") {                          echo "Mr. $name"                      } else {                          echo "Mrs. $name"                      }                  }              }          }     }  } |
|  |  |



****script**** block is wrapped inside the ****steps**** block. In the above example, we are printing the name passed as parameter suffixed with Mr. or Mrs. based on the Gender chosen.



We used ****build with parameters**** to pass params at the time of running the job.



## **environment**

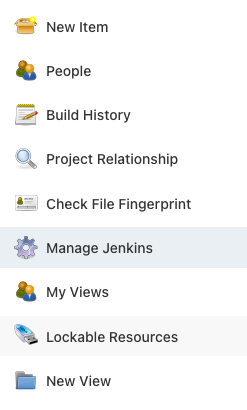
Key value pairs which helps passing values to job during job runtime from outside of Jenkinsfile. It’s one way of externalizing configuration.

****Example-**** Usually, [Jenkins jobs](https://www.lambdatest.com/blog/jenkins-freestyle-project/" \t "https://www.lambdatest.com/blog/jenkins-declarative-pipeline-examples/_blank) will run in a separate server. We may not be sure where the installation path of JAVA or JMeter is that server. Hence these are ideal candidates to be configured in Environment variables & passed during job run.

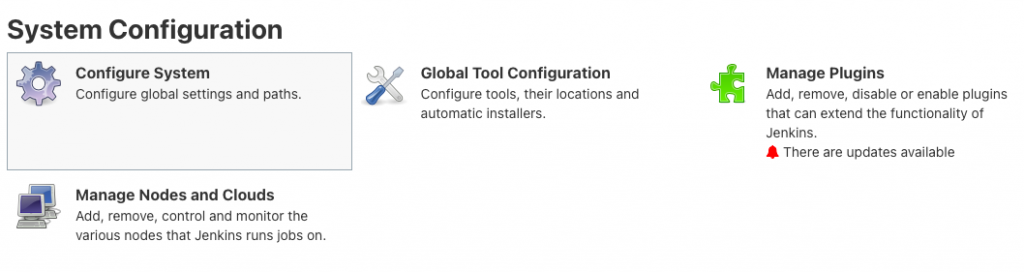
### Method 1 : Configure Environment Variable in Jenkins CI/CD portal

****Step 1:**** Open Jenkins Server URL (http://localhost:8080)

****Step 2:**** Click on Manage Jenkins from the left sidebar menu.



****Step 3:**** Click on ****Configure System**** under System Configuration.



****Step 4:**** Scroll down to the ****Global Properties**** section. This is where we will add our Environment variables.

****Step 5:**** Click on the ****Add**** button under Environment Variables & enter the Key & value.

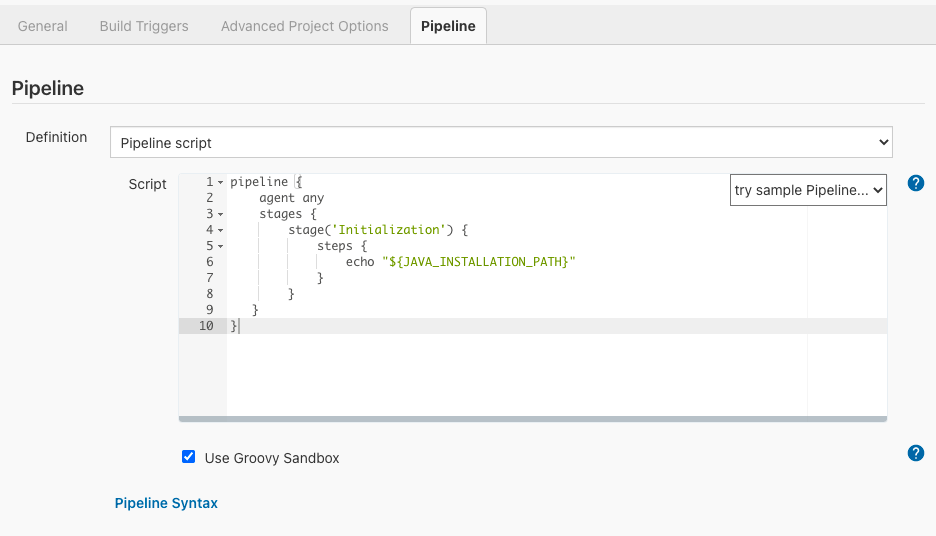
We have added the Java installation path under the variable name ****JAVA\_INSTALLATION\_PATH****



****Step 6:**** Click on ****Save****Button.

****Refer Environment variable in Jenkinsfile****  
We can refer to the Environment variables in declarative pipeline using the ${} syntax

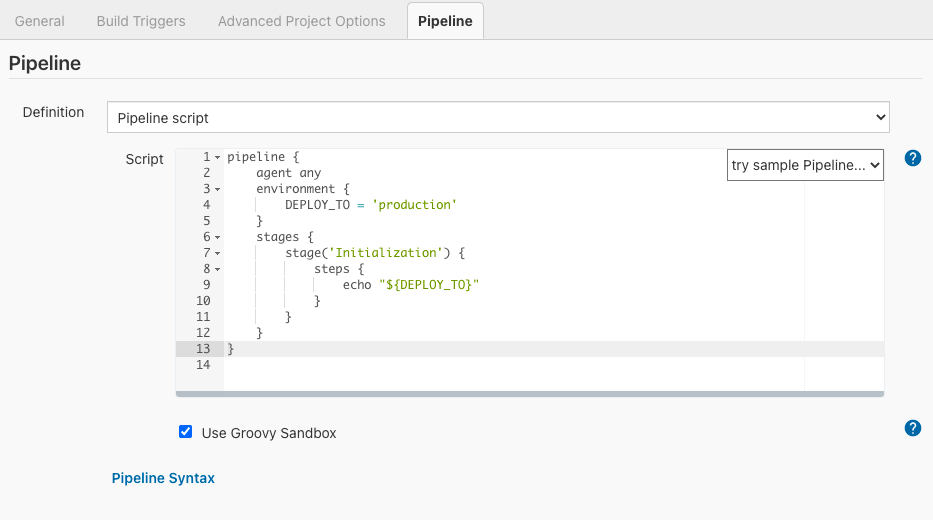
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | pipeline {      agent any      stages {          stage('Initialization') {              steps {                  echo "${JAVA\_INSTALLATION\_PATH}"              }          }     }  } |



### Method 2 : Creating & Referring Environment Variable in Jenkinsfile

We can create key value pairs of environment variables under ****environment**** block. It’s an ****optional**** block under pipeline.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | pipeline {      agent any      environment {          DEPLOY\_TO = 'production'      }      stages {          stage('Initialization') {              steps {                  echo "${DEPLOY\_TO}"              }          }      }  } |



### Method 3 : Initialize Environment variables using sh scripts in Jenkinsfile

Let’s say we need the timestamp when the job gets run for logging purposes. We can create an Environment variable which can hold the timestamp. But how to initialize it ?

We can use the shell script to fetch the current timestamp and assign it to the Environment variable.

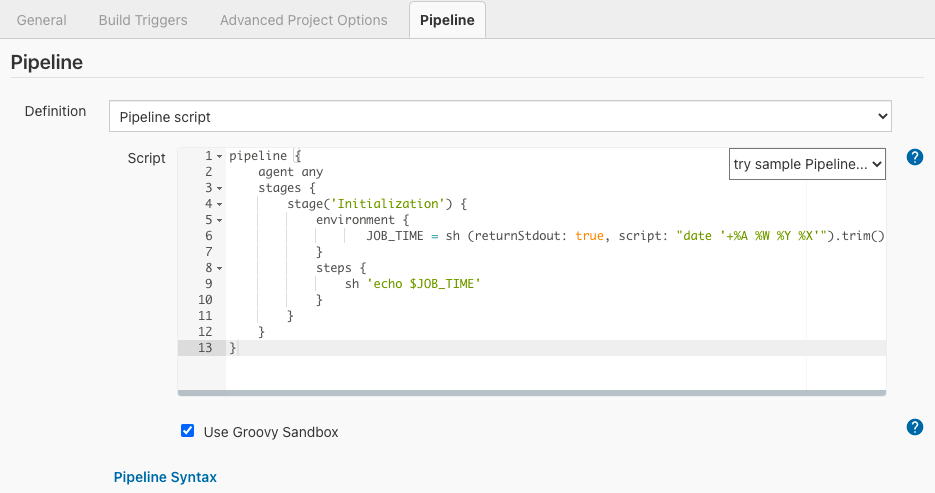
****Example-****

The following command will print date & time in shell

|  |  |
| --- | --- |
| 1  2 | >  date '+%A %W %Y %X'  Tuesday 03 2021 22:03:31 |

Lets see how to use the above command to initialize the Environment variable in Jenkinsfile.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | pipeline {      agent any      stages {          stage('Initialization') {              environment {                     JOB\_TIME = sh (returnStdout: true, script: "date '+%A %W %Y %X'").trim()              }              steps {                  sh 'echo $JOB\_TIME'              }          }      }  } |



****returnStdout:**** true makes sh step returning the output of the command so you can assign it to a variable.

### Load credentials via environment block

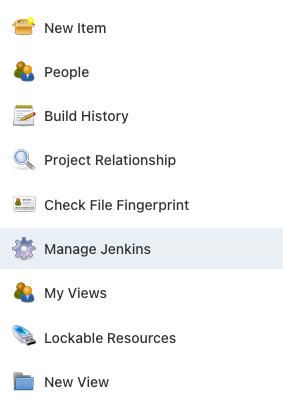
credentials() is a special method that can be used inside the environment block. This method helps loading the credentials defined at Jenkins configuration.

Lets see it with an example. First lets configure credential in Jenkins CI/CD-

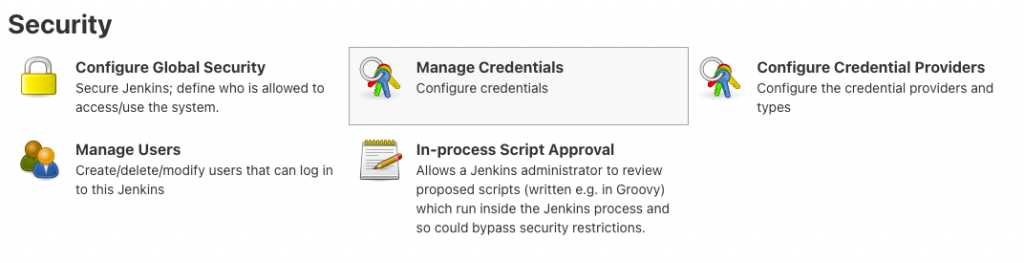
****Creating Credential in Jenkins portal****

****Step 1:**** Open Jenkins Server URL (http://localhost:8080)

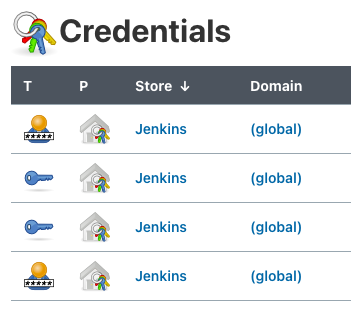
****Step 2:**** Click on ****Manage Jenkins**** from the left sidebar menu.



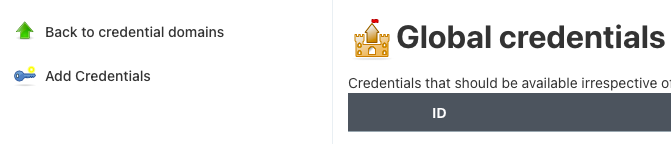
****Step 3:**** Click on ****Manage****Credentials under ****Security****.



****Step 4:**** Click on the ****Global****hyperlink.

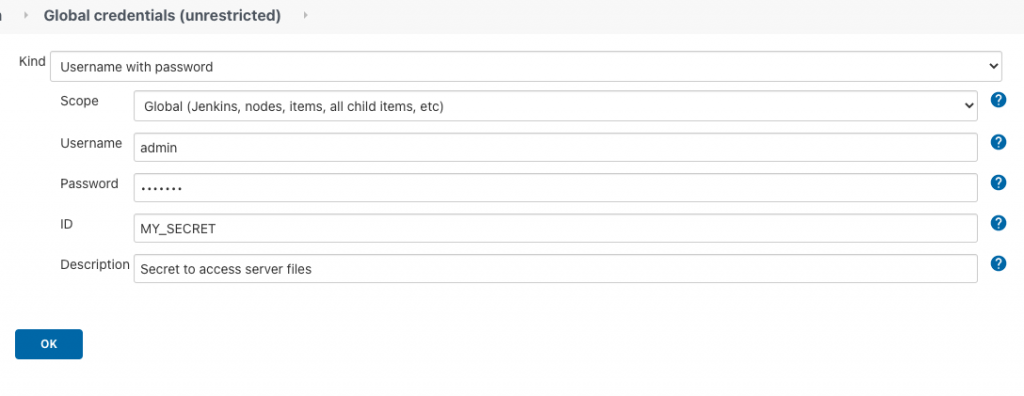


****Step 5:**** Click on Add ****Credentials****from the left side menu.



For demonstration, let me use the following values

* Kind – Username with password
* Scope – Global
* Username – admin
* Password – root123
* ID – MY\_SECRET
* Description – Secret to access server files



****Step 6:**** Click the ****OK**** button. Now our credentials are configured.

****Referring credential in Jenkinsfile****

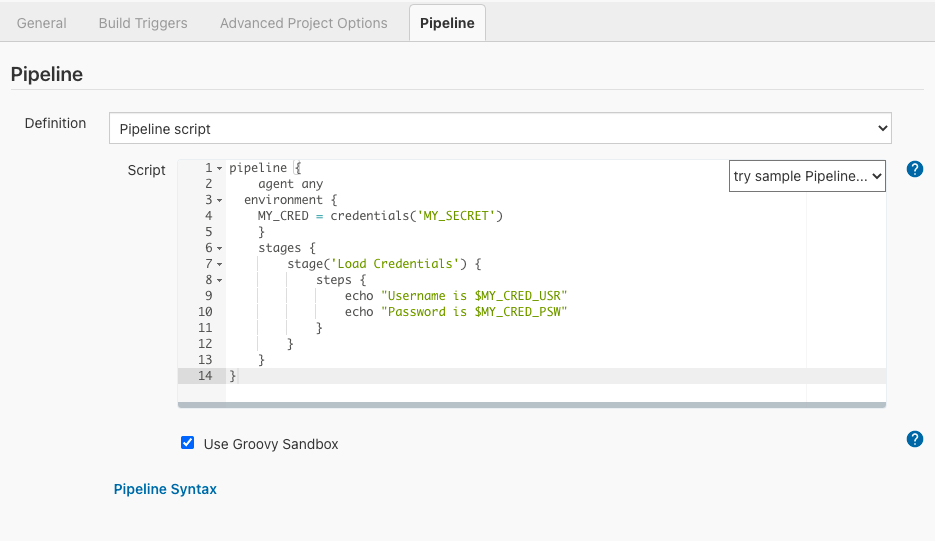
We can use the special method called ****credentials()**** to load credentials we configured. We need to use the ID we used while configuring credentials to load a particular credential.

****Example-**** credentials(‘MY\_SECRET’)

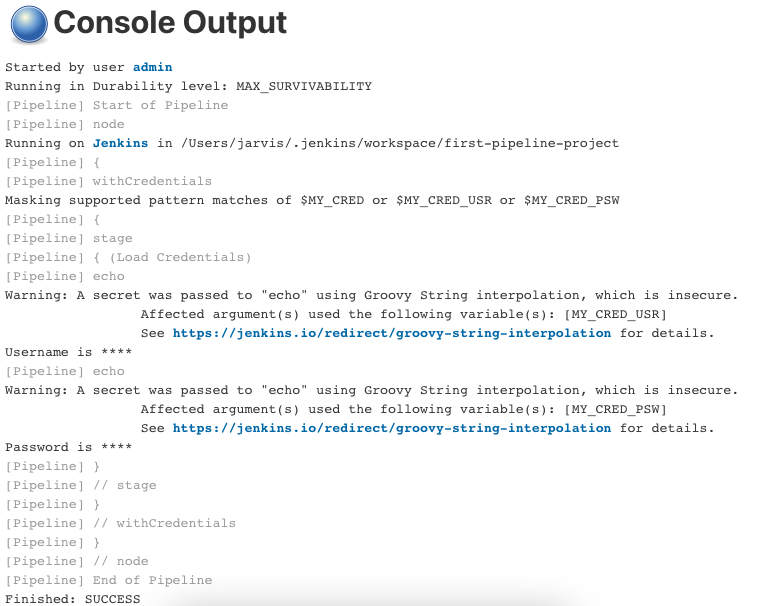
To refer to the username, append the word ****\_USR****to the variable we assigned credentials() output & append the word ****\_PSW****to get the password.

Example- Lets load the credential at ****MY\_CRED****variable using the ****credentials(‘MY\_SECRET’)**** method. We need to append ****\_USR ($MY\_CRED\_USR)**** &****\_PSW ($MY\_CRED\_PSW)**** to MY\_CRED variable to get the username & password.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | pipeline {      agent any    environment {     MY\_CRED = credentials('MY\_SECRET')      }      stages {          stage('Load Credentials') {              steps {                  echo "Username is $MY\_CRED\_USR"                  echo "Password is $MY\_CRED\_PSW"              }          }      }  } |



Interestingly, Jenkins print username & password as \*\*\*\* to avoid accidental leakage of credentials.

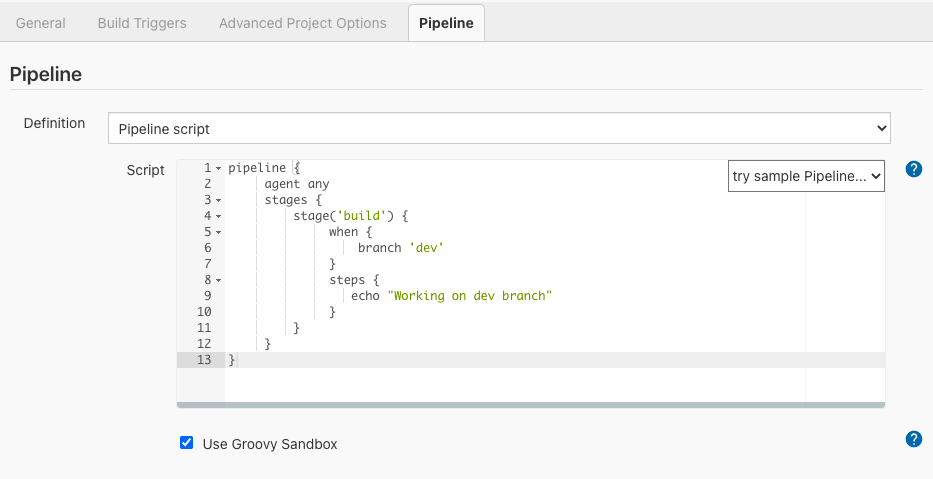


## **when**

Acts like if condition to decide whether to run the particular stage or not . Its an ****optional**** block.

1. ****when**** block

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | pipeline {       agent any       stages {           stage('build') {                when {                    branch 'dev'                }                steps {                   echo "Working on dev branch"                }           }       }  } |



1. when block using ****Groovy expression****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | pipeline {       agent any       stages {           stage('build') {                when {                    expression {                       return env.BRANCH\_NAME == 'dev';                    }                }                steps {                   echo "Working on dev branch"                }           }       }  } |



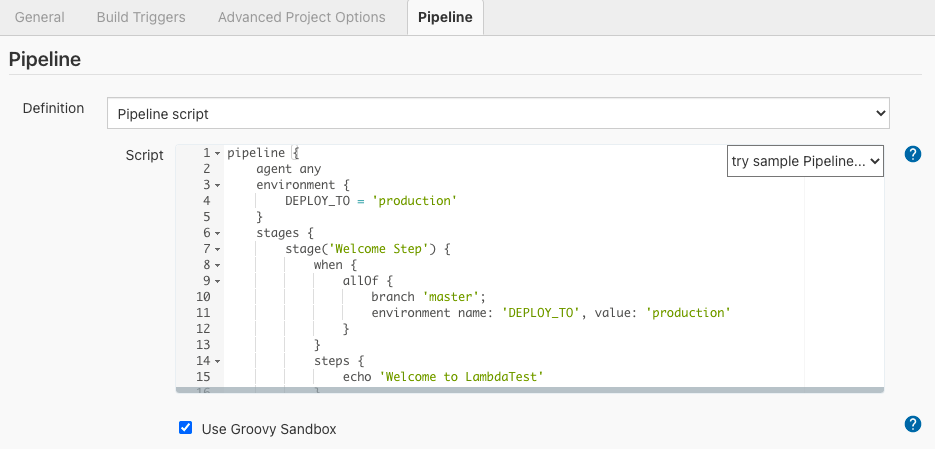
1. when block with ****environment variables****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | pipeline {      agent any      environment {          DEPLOY\_TO = 'production'      }      stages {          stage('Welcome Step') {               when {      environment name: 'DEPLOY\_TO', value: 'production'  }              steps {                  echo 'Welcome to LambdaTest'              }          }      }  } |



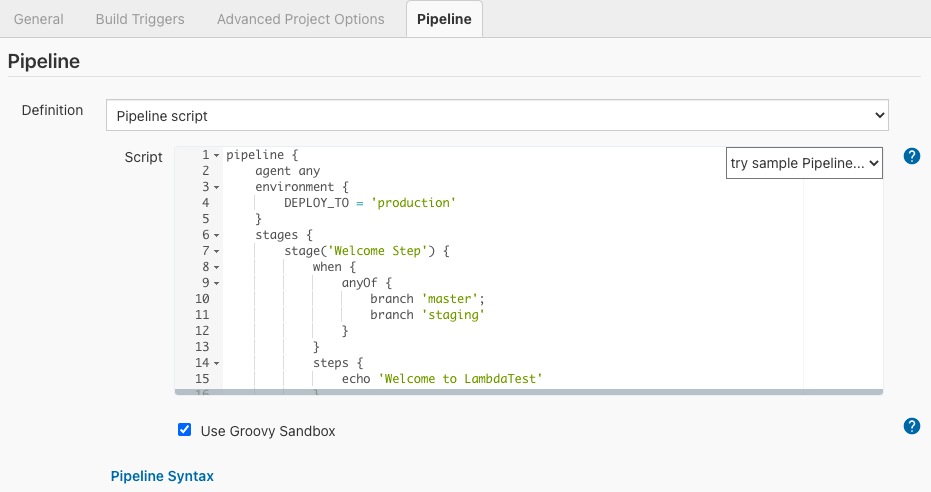
1. when block with ****multiple conditions & all conditions should be satisfied****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | pipeline {      agent any      environment {          DEPLOY\_TO = 'production'      }      stages {          stage('Welcome Step') {              when {                  allOf {                      branch 'master';                      environment name: 'DEPLOY\_TO', value: 'production'                  }              }              steps {                  echo 'Welcome to LambdaTest'              }          }      }  } |



1. when block with ****multiple conditions & any of the given conditions should be satisfied****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | pipeline {      agent any      environment {          DEPLOY\_TO = 'production'      }      stages {          stage('Welcome Step') {              when {                  anyOf {                      branch 'master';                      branch 'staging'                  }              }              steps {                  echo 'Welcome to LambdaTest'              }          }      }  } |



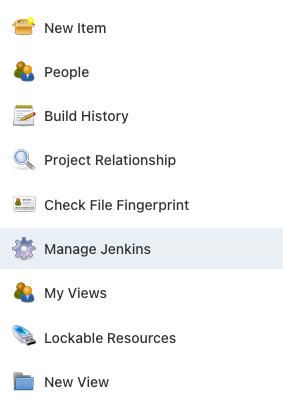
## **tools**

This block lets us add pre configured tools like Maven or Java to our job’s PATH. It’s an ****optional**** block.

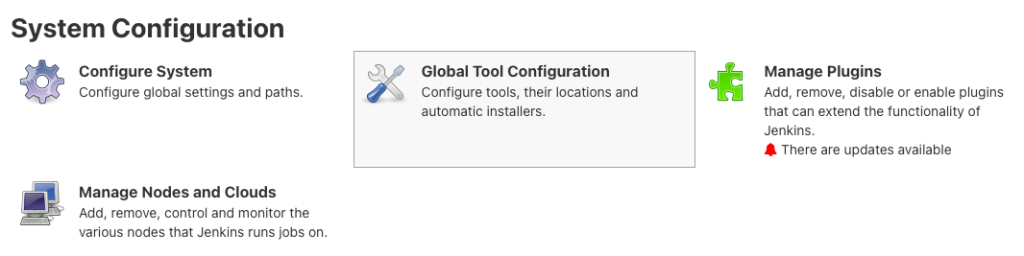
To use any tool, they have to be pre-configured under the ****Global Tools Configuration**** section. For this example, let’s see how to configure Maven.

****Step 1:**** Open Jenkins Server URL (http://localhost:8080)

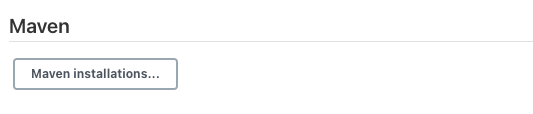
****Step 2:**** Click on ****Manage Jenkins**** from the left sidebar menu.



****Step 3****: Click on ****Global Tools Configuration**** under System Configuration.



****Step 4:**** Scroll down to the ****Maven**** section & click on ****Maven Installations****.



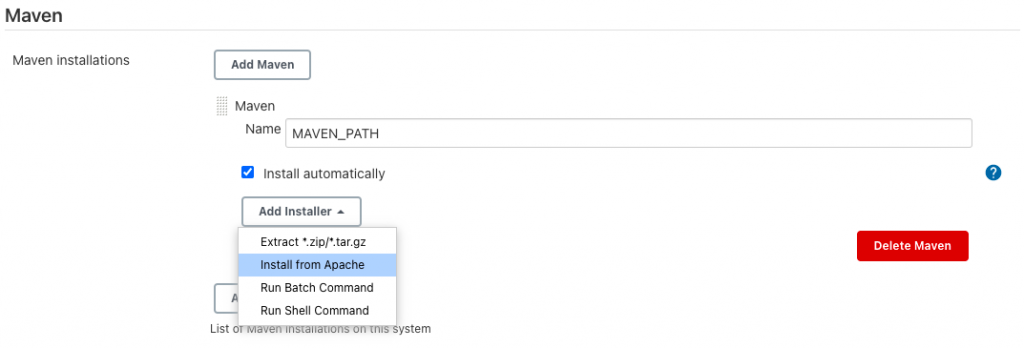
****Step 5:**** Click on the ****Add Maven****button. Enter the following values in the configuration.

* Name – Maven 3.5.0
* MAVEN\_HOME – Enter the maven installation path in local

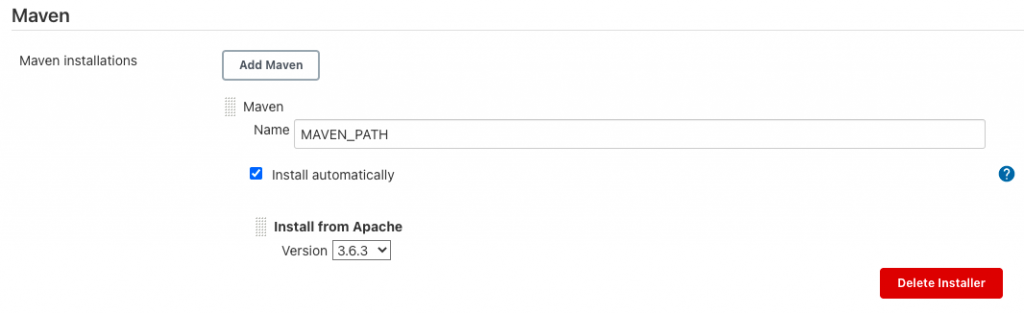


Alternatively we can download Maven from Internet instead of pointing to the local installation path by enabling the Install Automatically checkbox.

****Step 6:**** Click on ****Add Installer**** button & choose ****Install from Apache****.



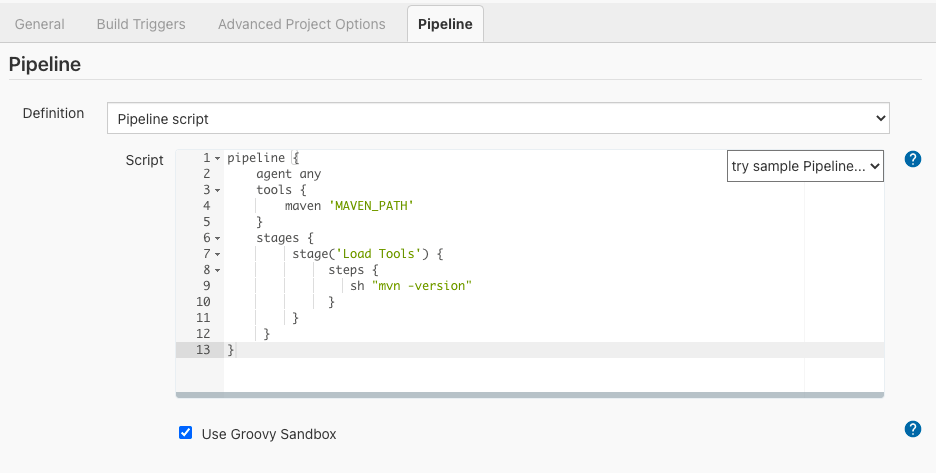
****Step 7:**** Choose the Maven version to download. We have chosen the Maven version ****3.6.3****.



****Step 8:**** Click on the ****Save****button.

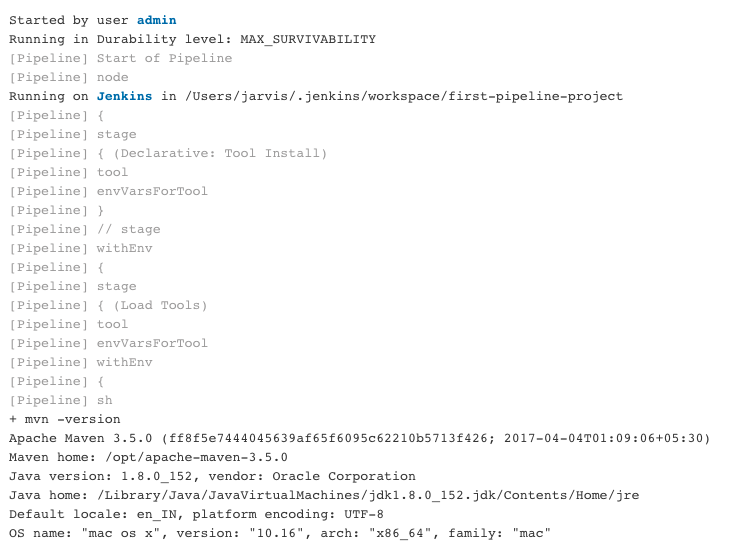
Refer to Maven tool in Pipeline in tools block with the name we used while configuring in Global Tools Configuration (MAVEN\_PATH in this example).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | pipeline {      agent any      tools {          maven 'MAVEN\_PATH'      }      stages {           stage('Load Tools') {                steps {                   sh "mvn -version"                }           }       }  } |



****Console Output-****

Run the job & we should see the mvn -version command response in console output-



Similarly we can also configure & use tools like Java, Gradle, Git etc from tools block in pipeline code.

## **parallel**

parallel blocks allow us to ****run multiple stage blocks concurrently****. It’s ideal to parallelize stages which can run independently. We can define agents for each stage within a parallel block, so each stage will run on its own agent.

****Example-**** When we want to run a piece of script in windows agent & another script in linux agent as part of the build, we can make them run concurrently using parallel blocks.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | pipeline {      agent any      stages {             stage('Parallel Stage') {                parallel {                  stage('windows script') {                      agent {                          label "windows"                      }                      steps {                           echo "Running in windows agent"  bat 'echo %PATH%'                      }                  }                  stage('linux script') {                      agent {                          label "linux"                      }                      steps {                         sh "Running in Linux agent"                      }                  }               }          }      }  } |

## **post**

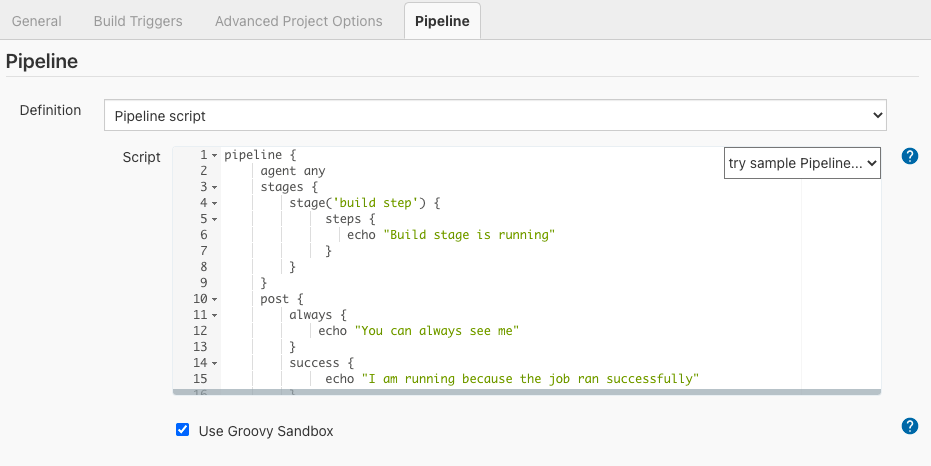
****post****block contains additional actions to be performed on completion of the pipeline execution. It can contain one or many of the following conditional blocks – always, changed, aborted, failure, success, unstable etc.

****post block conditions****

****always –**** run this post block irrespective of the pipeline execution status  
****changed –**** run this post block only if the pipeline’s execution status is different from the previous build run. E.g., the build failed at an earlier run & ran successfully this time.  
****aborted –**** if the build is aborted in the middle of the run. Usually due to manual stopping of the build run  
****failure –**** if the build status is failure  
****success –****if the build ran successfully  
****unstable –**** build is successful but not healthy. E.g. On a particular build run, test cases ran successfully, but test coverage % is less than expected, then the build can be marked as unstable

****Example-****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | pipeline {       agent any       stages {           stage('build step') {                steps {                   echo "Build stage is running"                }           }       }       post {           always {               echo "You can always see me"           }           success {                echo "I am running because the job ran successfully"           }           unstable {                echo "Gear up ! The build is unstable. Try fix it"           }           failure {               echo "OMG ! The build failed"           }       }  } |

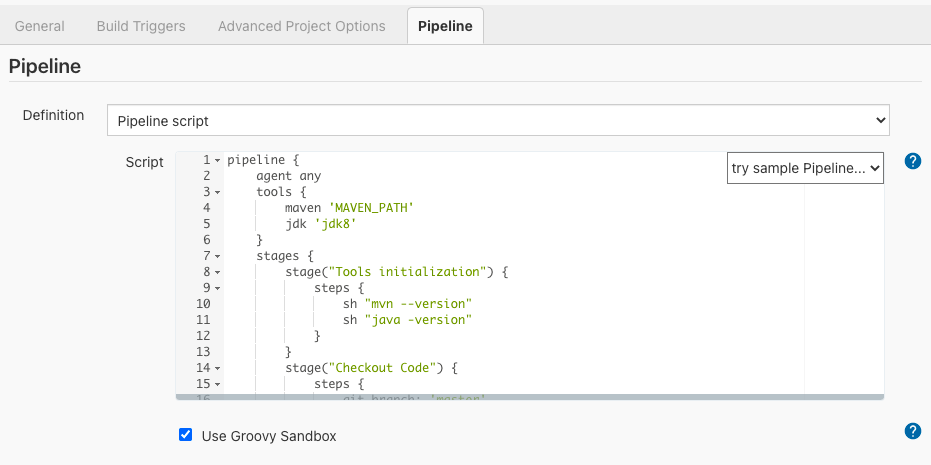


## **Marking build as Unstable**

There are scenarios where we don’t want to mark build as failure but want to mark build as unstable.

****Example-**** When test case coverage doesn’t cross the threshold we expect, we can mark the build as ****UNSTABLE****.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | pipeline {      agent any      tools {          maven 'MAVEN\_PATH'          jdk 'jdk8'      }      stages {          stage("Tools initialization") {              steps {                  sh "mvn --version"                  sh "java -version"              }          }          stage("Checkout Code") {              steps {                  git branch: 'master',                  url: "https://github.com/iamvickyav/spring-boot-data-H2-embedded.git"              }          }          stage("Building Application") {              steps {                 sh "mvn clean package"              }          }          stage("Code coverage") {             steps {                 jacoco(                      execPattern: '\*\*/target/\*\*.exec',                      classPattern: '\*\*/target/classes',                      sourcePattern: '\*\*/src',                      inclusionPattern: 'com/iamvickyav/\*\*',                      changeBuildStatus: true,                      minimumInstructionCoverage: '30',                      maximumInstructionCoverage: '80')                 }             }          }  } |



Sample can be found here at [GitHub](https://github.com/iamvickyav/jenkins-declarative-pipeline-notes/blob/master/declarative-syntax-samples.md" \l "mark-build-as-unstable-based-on-code-coverage" \t "https://www.lambdatest.com/blog/jenkins-declarative-pipeline-examples/_blank).

****jacoco()**** is a special method which helps us configure Jacoco reports including the minimum & maximum coverage threshold. In the above example, min coverage (minimumInstructionCoverage) check is set as 30 & max coverage (maximumInstructionCoverage) check is set as 80.

So if the code coverage is-

****Less than 30 –**** Build will be marked as Failure.  
****Between 30 & 80 –**** Build will be marked as Unstable.  
****Above 80 –**** Build will be marked as Success.

## **triggers**

Instead of triggering the build manually, we can configure build to run in certain time intervals using the triggers block. We can use the special method called cron() within triggers block to configure the build schedule.

****Understanding cron****

Cron configuration contains 5 fields representing minute (0-59), hour (0-23), day of the month (1-31), month (1-12), day of the week (0-7)

****Example-****

****Every 15 minutes – H/15 \* \* \* \*  
Every 15 minutes but only between Monday & Friday – H/15 \* \* \* 1-5****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | pipeline {      agent any      triggers {          cron('H/15 \* \* \* \*')      }      stages {          stage('Example') {              steps {                  echo 'Hello World'              }          }      }  } |

