**Jenkins Pipeline: A Beginner’s Guide To Continuous Delivery**

Jenkins for the continuous delivery pipeline, you can interpret the demand for***Continuous delivery & Jenkins skills***. Have you ever wondered why Jenkins has gained so much popularity, especially over the recent years? One of the major factors that contribute to it’s popularity is the **Jenkins pipeline** and if you’re looking for a simple Jenkins pipeline tutorial, this blog is your go-to. Jenkins pipeline is a continuous delivery pipeline that executes the software workflow as code.

List of the topics covered:

* What is a Jenkins pipeline?
* What is a Jenkinsfile?
* Pipeline concepts
* Creating your first Jenkins pipeline
* Declarative pipeline demo
* Scripted pipeline demo

## ****Jenkins Pipeline Tutorial****

We’re all aware that Jenkins has proven to be an expert in implementing continuous integration, continuous testing and continuous deployment to produce good quality software. When it comes to **continuous delivery**, Jenkins uses a feature called Jenkins pipeline. In order to understand why Jenkins pipeline was introduced, we have to understand what continuous delivery is and why it is important.



Continuous delivery is the capability to release a software at all times. It is a practice which ensures that the software is always in a **production-ready state**.

What does this mean? It means that every time a change is made to the code or the infrastructure, the software team must work in such a way that these changes are built quickly and tested using various **automation tools**after which the build is subjected to production.

By speeding up the delivery process, the development team will get more time to implement any required **feedback**. This process, of getting the software from the build to the production state at a faster rate is carried out by implementing continuous integration and continuous delivery.

Continuous delivery ensures that the software is built, tested and released more frequently. It reduces the cost, time and risk of the incremental software releases. To carry out continuous delivery, Jenkins introduced a new feature called Jenkins pipeline. This Jenkins pipeline tutorial will help you understand the importance of a Jenkins pipeline.

## ****What is a Jenkins pipeline?****

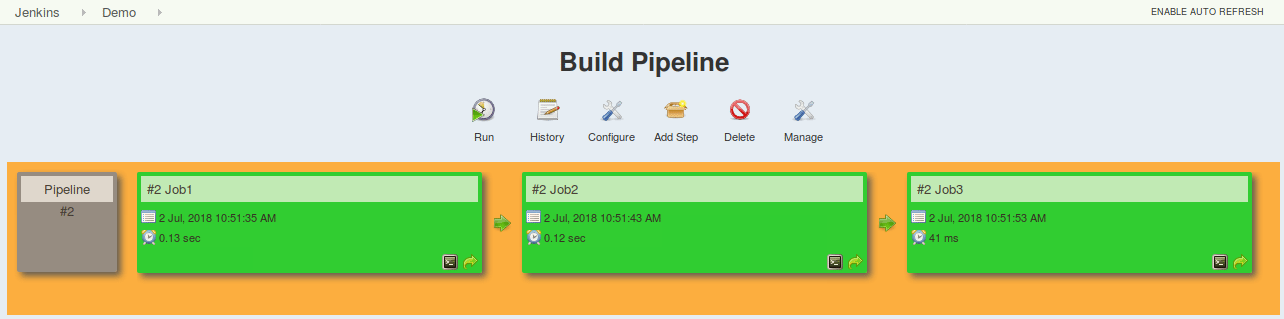
A pipeline is a collection of jobs that brings the software from version control into the hands of the end users by using automation tools. It is a feature used to **incorporate continuous delivery** in our software development workflow.

Over the years, there have been multiple Jenkins pipeline releases including, Jenkins Build flow, Jenkins Build Pipeline plugin, Jenkins Workflow, etc. What are the key features of these plugins?

* They represent multiple Jenkins jobs as one whole workflow in the form of a pipeline.
* What do these pipelines do? These pipelines are a **collection of Jenkins jobs** which trigger each other in a specified sequence.

Let me explain this with an example. Suppose I’m developing a small application on Jenkins and I want to build, test and deploy it. To do this, I will allot 3 jobs to perform each process. So, job1 would be for build, job2 would perform tests and job3 for deployment. I can use the Jenkins build pipeline plugin to perform this task. After creating three jobs and chaining them in a sequence, the build plugin will run these jobs as a pipeline.

This image shows a view of all the 3 jobs that run concurrently in the pipeline.



Build Pipeline Plugin output – Jenkins Pipeline Tutorial

This approach is effective for deploying small applications. But what happens when there are complex pipelines with several processes (build, test, unit test, integration test, pre-deploy, deploy, monitor) running 100’s of jobs?

The maintenance cost for such a complex pipeline is huge and increases with the number of processes. It also becomes tedious to build and manage such a vast number of jobs. To overcome this issue, a new feature called **Jenkins Pipeline Project** was introduced.

The key feature of this pipeline is to define the entire deployment flow through code. What does this mean? It means that all the standard jobs defined by Jenkins are manually written as one whole script and they can be stored in a version control system. It basically follows the ‘**pipeline as code**’ discipline. Instead of building several jobs for each phase, you can now code the entire workflow and put it in a **Jenkinsfile**. Below is a list of reasons why you should use the Jenkins Pipeline.

### ****Jenkins Pipeline Advantages****

* It models simple to complex pipelines as code by using **Groovy DSL** (Domain Specific Language)
* The code is stored in a text file called the Jenkinsfile which can be **checked into a SCM** (Source Code Management)
* Improves user interface by incorporating **user****input** within the pipeline
* It is durable in terms of unplanned restart of the Jenkins master
* It can restart from saved **checkpoints**
* It supports complex pipelines by incorporating conditional loops, fork or join operations and allowing tasks to be performed in parallel
* It can integrate with several other plugins

## ****What is a Jenkinsfile?****

A Jenkinsfile is a text file that stores the entire workflow as code and it can be checked into a SCM on your local system. How is this advantageous? This enables the developers to **access, edit and check the code at all times**.

The Jenkinsfile is written using the Groovy DSL and it can be created through a text/groovy editor or through the configuration page on the Jenkins instance. It is written based on two syntaxes, namely:

### ****Declarative pipeline syntax****

### ****Scripted pipeline syntax****

Declarative pipeline is a relatively new feature that supports the pipeline as code concept. It makes the pipeline code easier to read and write. This code is written in a Jenkinsfile which can be checked into a source control management system such as Git.

Whereas, the scripted pipeline is a traditional way of writing the code. In this pipeline, the Jenkinsfile is **written on the Jenkins UI instance**. Though both these pipelines are based on the groovy DSL, the scripted pipeline uses stricter groovy based syntaxes because it was the first pipeline to be built on the groovy foundation. Since this Groovy script was not typically desirable to all the users, the declarative pipeline was introduced to offer a simpler and more optioned Groovy syntax.

The declarative pipeline is defined within a block labelled ‘pipeline’ whereas the scripted pipeline is defined within a ‘**node**’. This will be explained below with an example.

## ****Pipeline concepts****

### ****Pipeline****

This is a user defined block which contains all the processes such as build, test, deploy, etc. It is a collection of all the stages in a Jenkinsfile. All the stages and steps are defined within this block. It is the key block for a declarative pipeline syntax.



### ****Node****

A node is a machine that executes an entire workflow. It is a key part of the scripted pipeline syntax.



There are various mandatory sections which are common to both the declarative and scripted pipelines, such as stages, agent and steps that must be defined within the pipeline. These are explained below:

### ****Agent****

An agent is a directive that can run multiple builds with only one instance of Jenkins. This feature helps to distribute the workload to different agents and execute several projects within a single Jenkins instance. It instructs Jenkins to **allocate an executor** for the builds.

A single agent can be specified for an entire pipeline or specific agents can be allotted to execute each stage within a pipeline. Few of the parameters used with agents are:

##### **Any**

Runs the pipeline/ stage on any available agent.

##### **None**

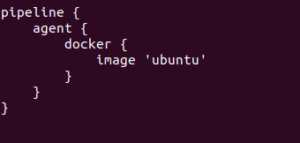
This parameter is applied at the root of the pipeline and it indicates that there is no global agent for the entire pipeline and each stage must specify its own agent.

##### **Label**

Executes the pipeline/stage on the labelled agent.

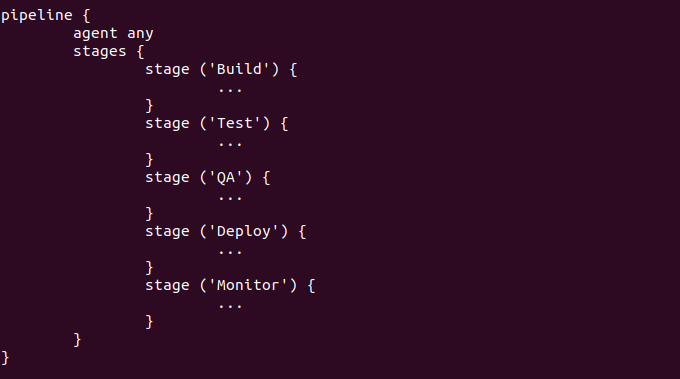
##### **Docker**

This parameter uses docker container as an execution environment for the pipeline or a specific stage. In the below example I’m using docker to pull an ubuntu image. This image can now be used as an execution environment to run multiple commands.



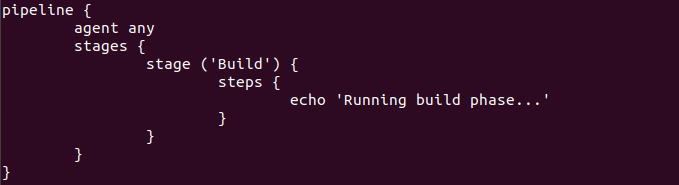
### ****Stages****

This block contains all the work that needs to be carried out. The work is specified in the form of stages. There can be more than one stage within this directive. Each stage performs a specific task. In the following example, I’ve created multiple stages, each performing a specific task.



### ****Steps****

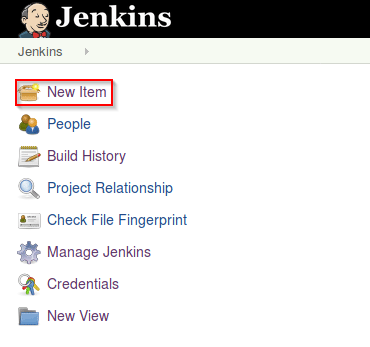
A series of steps can be defined within a stage block. These steps are carried out in sequence to execute a stage. There must be at least one step within a steps directive. In the following example I’ve implemented an echo command within the build stage. This command is executed as a part of the ‘Build’ stage.



Now that you are familiar with the basic pipeline concepts let’s start of with the Jenkins pipeline tutorial. Firstly, let’s learn how to create a Jenkins pipeline.

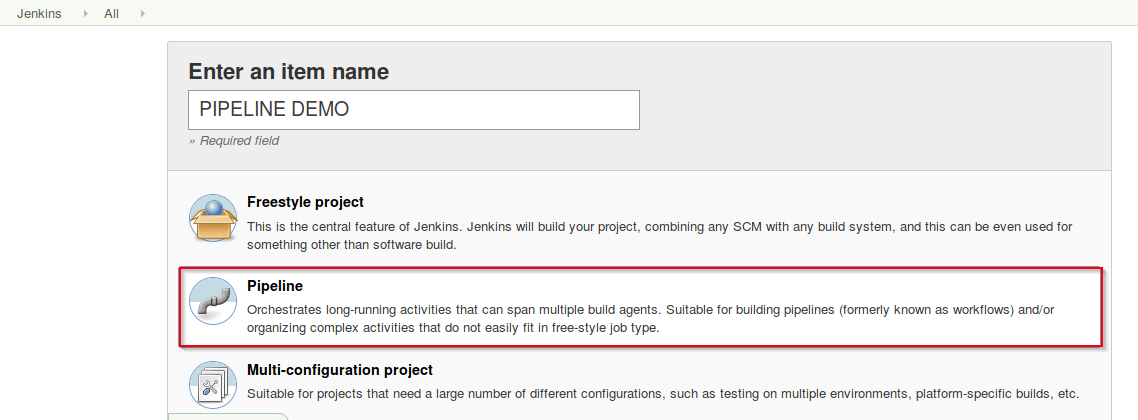
### ****Creating your first Jenkins pipeline.****

**Step 1**: Log into Jenkins and select ‘New item’ from the dashboard.

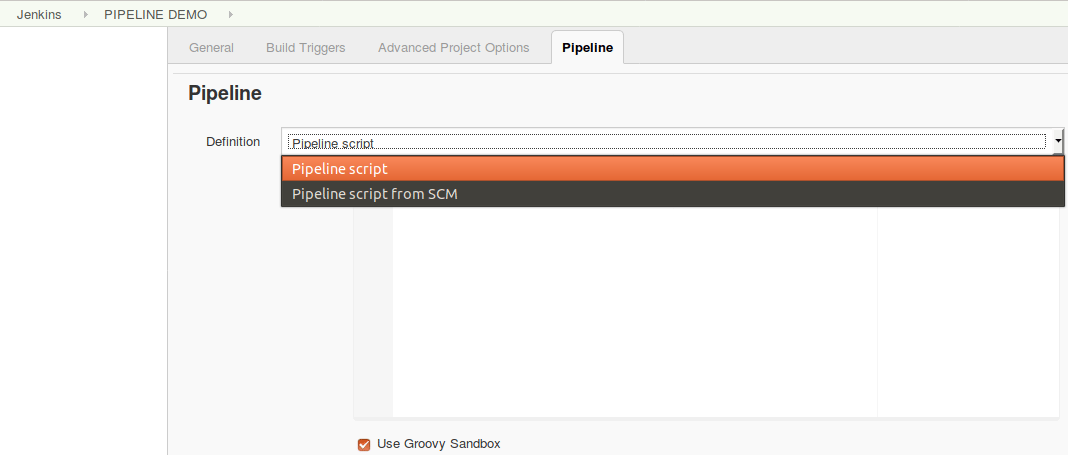


Jenkins Dashboard – Jenkins Pipeline Tutorial

**Step 2**: Next, enter a name for your pipeline and select ‘pipeline’ project. Click on ‘ok’ to proceed.

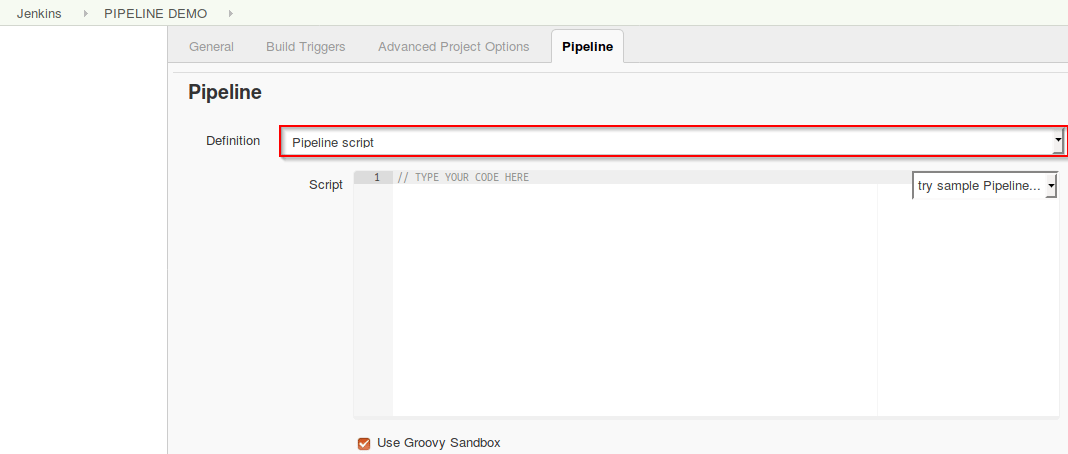
Enter the project name – Jenkins Pipeline Tutorial

**Step 3**: Scroll down to the pipeline and choose if you want a declarative pipeline or a scripted one.



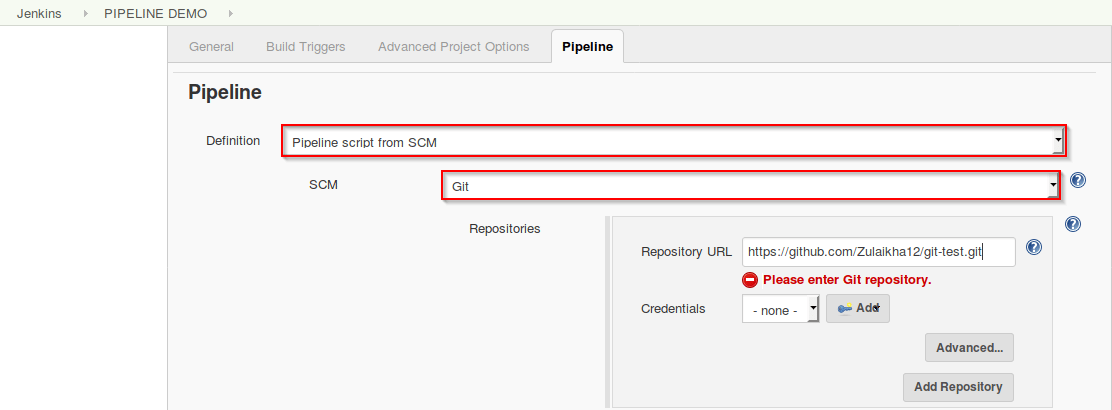
Declarative or scripted pipeline – Jenkins Pipeline Tutorial

**Step 4a**: If you want a scripted pipeline then choose ‘pipeline script’ and start typing your code.

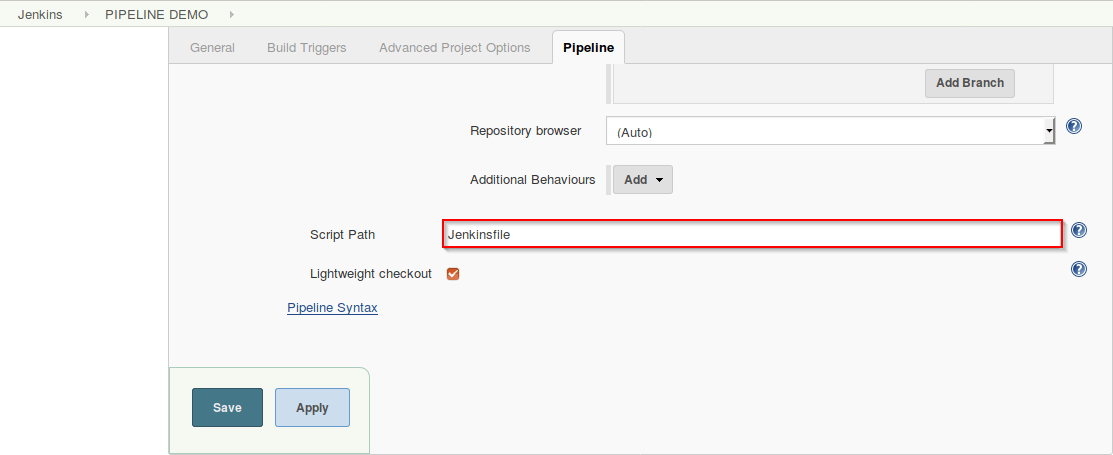


Scripted Pipeline – Jenkins Pipeline Tutorial

**Step 4b**: If you want a declarative pipeline then select ‘pipeline script from SCM’ and choose your SCM. In my case I’m going to use Git throughout this demo. Enter your repository URL.

Declarative pipeline – Jenkins Pipeline Tutorial

**Step 5**: Within the script path is the name of the Jenkinsfile that is going to be accessed from your SCM to run. Finally click on ‘apply’ and ‘save’. You have successfully created your first Jenkins pipeline.

Script path – Jenkins Pipeline Tutorial

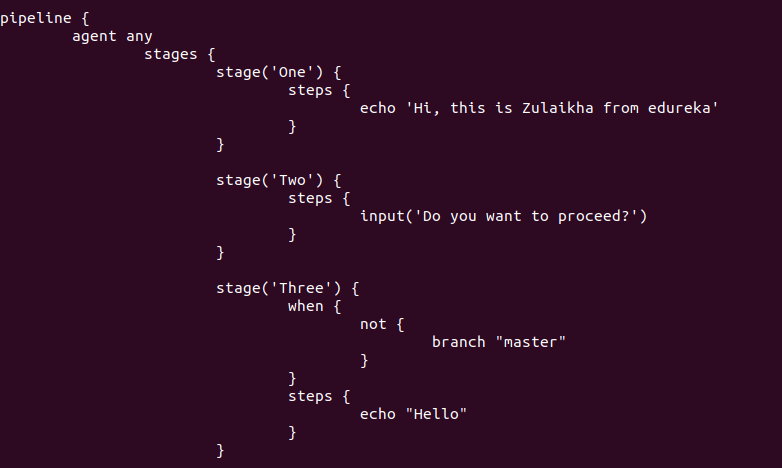
Now that you know how to create a pipeline, lets get started with the demo.

### ****Declarative Pipeline Demo****

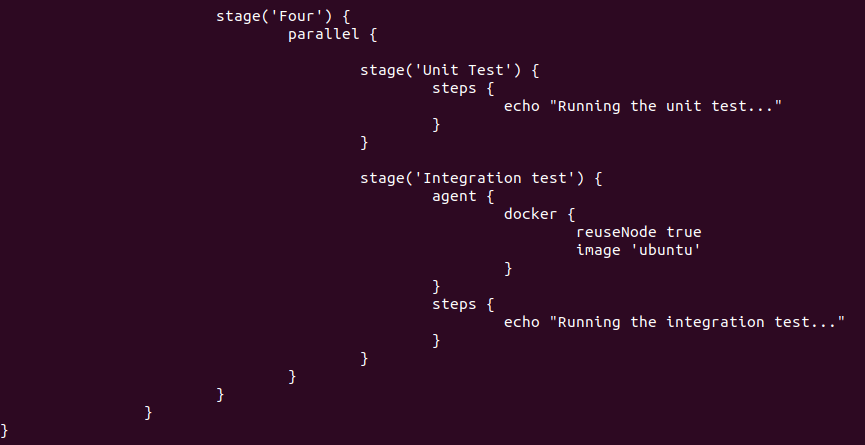
The first part of the demo shows the working of a declarative pipeline. Refer the above ‘Creating your first Jenkins pipeline’ to start. Let me start the demo by explaining the code I’ve written in my Jenkinsfile.

Since this is a declarative pipeline, I’m writing the code locally in a file named ‘Jenkinsfile’ and then pushing this file into my global git repository. While executing the ‘Declarative pipeline’ demo, this file will be accessed from my git repository. The following is a simple demonstration of building a pipeline to run multiple stages, each performing a specific task.

* The declarative pipeline is defined by writing the code within a pipeline block. Within the block I’ve defined an agent with the tag ‘any’. This means that the pipeline is run on any available executor.
* Next, I’ve created four stages, each performing a simple task.
* Stage one executes a simple echo command which is specified within the ‘steps’ block.
* Stage two executes an input directive. This directive allows to **prompt a user input** in a stage. It displays a message and waits for the user input. If the input is approved, then the stage will trigger further deployments.
* In this demo a simple input message ‘Do you want to proceed?’ is displayed. On receiving the user input the pipeline either proceeds with the execution or aborts.



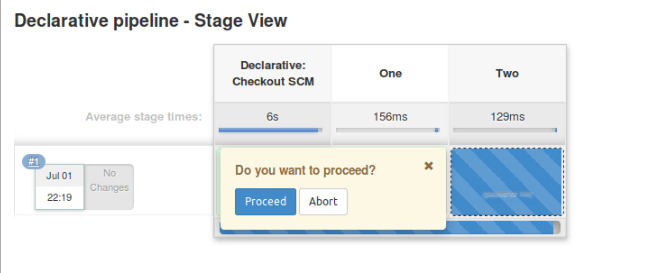
* Stage three runs a ‘when’ directive with a ‘not’ tag. This directive allows you to execute a step depending on the **conditions defined** within the ‘when’ loop. If the conditions are met, the corresponding stage will be executed. It must be defined at a stage level.
* In this demo, I’m using a ‘not’ tag. This tag executes a stage when the nested condition is **false**. Hence when the ‘branch is master’ holds false, the echo command in the following step is executed.



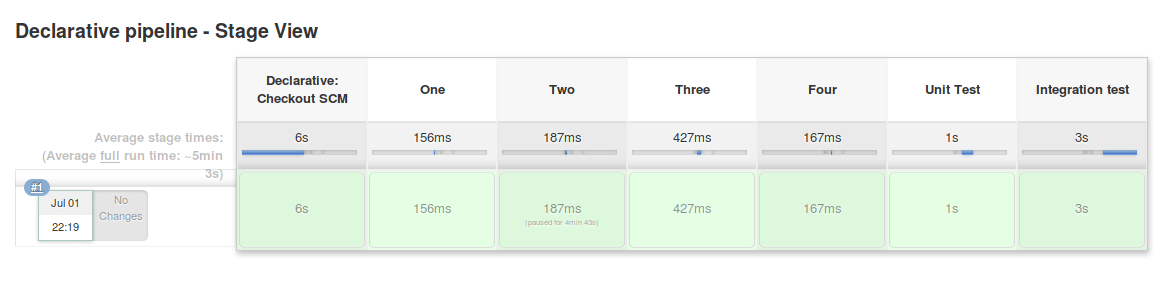
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | pipeline {           agent any           stages {                   stage('One') {                   steps {                       echo 'Hi, this is Zulaikha from vision'                   }                   }                   stage('Two') {                   steps {                      input('Do you want to proceed?')                   }                   }                   stage('Three') {                   when {                         not {                              branch "master"                         }                   }                   steps {                         echo "Hello"                   }                   }                   stage('Four') {                   parallel {                              stage('Unit Test') {                             steps {                                  echo "Running the unit test..."                             }                             }                              stage('Integration test') {                                agent {                                      docker {                                              reuseNode true                                              image 'ubuntu'                                             }                                      }                                steps {                                  echo "Running the integration test..."                                }                             }                             }                             }                }  } |

* Stage four runs a parallel directive. This directive allows you to run nested stages in parallel. Here, I’m running two nested stages in parallel, namely, ‘Unit test’ and ‘Integration test’. Within the integration test stage, I’m defining a stage specific docker agent. This docker agent will execute the ‘Integration test’ stage.
* Within the stage are two commands. The**reuseNode** is a Boolean and on returning true, the docker container would run on the agent specified at the top-level of the pipeline, in this case the agent specified at the top-level is ‘any’ which means that the container would be executed on any available node. By default this Boolean returns false.
* There are some restrictions while using the parallel directive:
  + A stage can either have a parallel or steps block, **but not both**
  + Within a parallel directive you cannot nest another parallel directive
  + If a stage has a parallel directive then you cannot define ‘agent’ or ‘tool’ directives

Now that I’ve explained the code, lets run the pipeline. The following screenshot is the result of the pipeline. In the below image, the pipeline waits for the user input and on clicking ‘proceed’, the execution resumes.



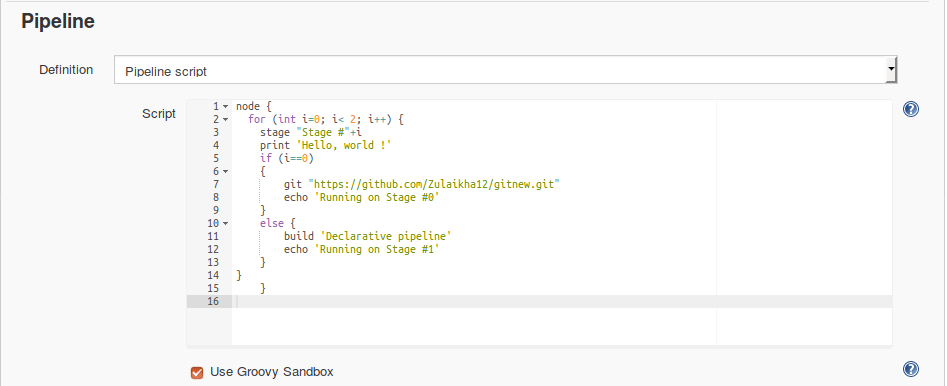
Waiting for user input – Jenkins Pipeline Tutorial



                                                             Final output – Jenkins Pipeline Tutorial

### ****Scripted Pipeline Demo****

To give you a basic understanding of the scripted pipeline, lets execute a simple code. Refer to Creating your first Jenkins pipeline to create the scripted pipeline. I will run the following script.



Pipeline script – Jenkins Pipeline Tutorial

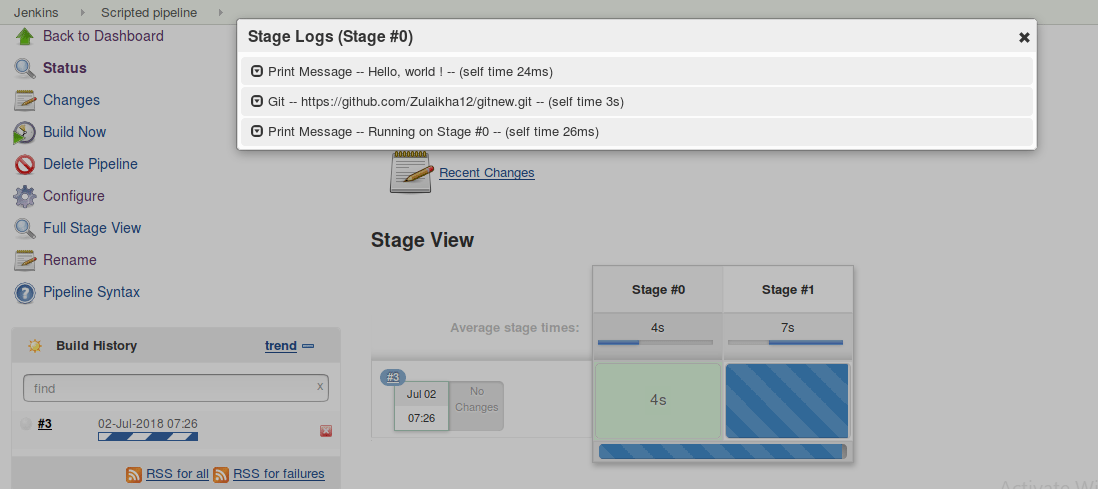
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | node {        for (i=0; i<2; i++) {             stage "Stage #"+i             print 'Hello, world !'             if (i==0)             {                 git "https://github.com/Zulaikha12/gitnew.git"                 echo 'Running on Stage #0'             }             else {                 build 'Declarative pipeline'                 echo 'Running on Stage #1'             }        }  } |

In the above code I have defined a ‘node’ block within which I’m running the following:

* The conditional ‘for’ loop. This for loop is for creating 2 stages namely, Stage #0 and Stage #1. Once the stages are created they print the ‘hello world!’ message
* Next, I’m defining a simple ‘if else’ statement. If the value of ‘i’ equals to zero, then stage #0 will execute the following commands (git and echo). A ‘git’ command is used to clone the specified git directory and the echo command simply displays the specified message
* The else statement is executed when ‘i’ is not equal to zero. Therefore, stage #1 will run the commands within the else block. The ‘build’ command simply runs the job specified, in this case it runs the ‘Declarative pipeline’ that we created earlier in the demo. Once it completes the execution of the job, it runs the echo command

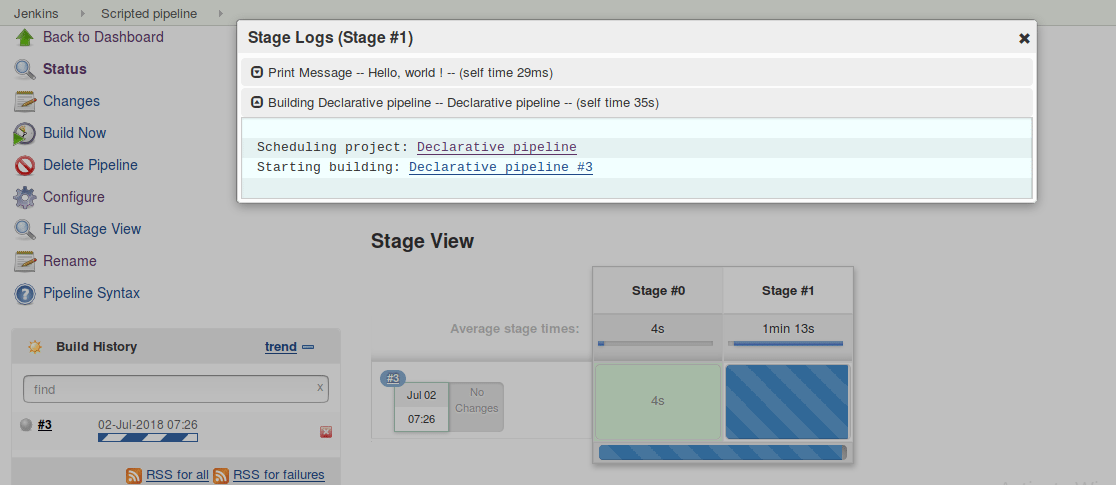
Now that I’ve explained the code, lets run the pipeline. The following screenshot is the result of the Scripted pipeline.

1. Shows the results of Stage #0



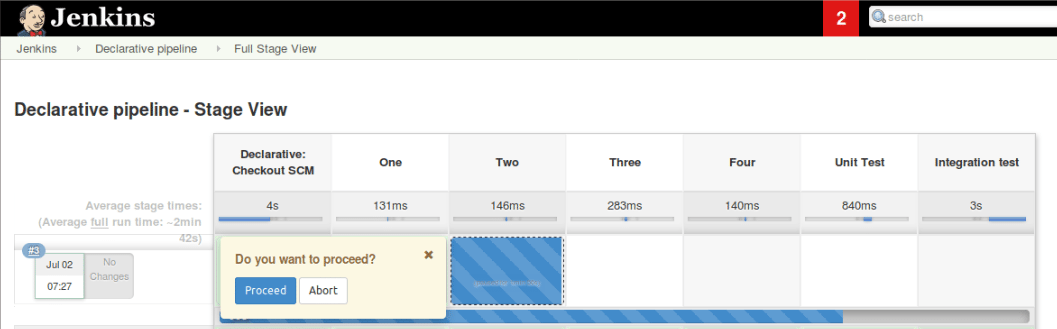
                                                                         Stage #0 – Jenkins Pipeline Tutorial

2. Shows the logs of Stage #1 and starts building the ‘Declarative pipeline’



Stage #1 running the declarative pipeline – Jenkins Pipeline Tutorial

3. Execution of the ‘Declarative pipeline’ job.



                                                             Execution of declarative pipeline-Jenkins Pipeline Tutorial

4. Results.

