Creating Pipeline using 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:

1.Declarative pipeline syntax

2.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.

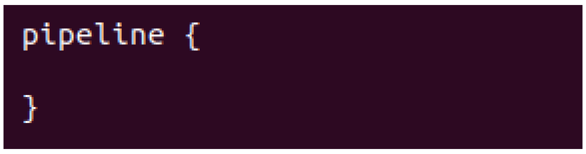
Whereas, the scripted pipeline is a traditional way of writing the code. In this pipeline, the Jenkinsfile is written on the Jenkins UI instance. Though both these pipelines are based on the groovy DSL, the scripted pipeline uses stricter groovy based syntaxes because it was the first pipeline to be 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

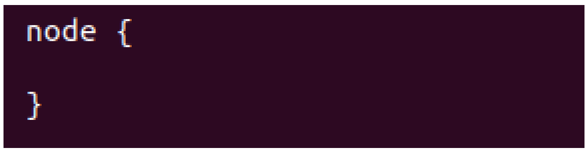
This is a user defined block which contains all the processes such as build, test, deploy, etc. It is a collection of all the stages in a Jenkinsfile. All the stages and steps are defined within this block. It is the key block for a declarative pipeline syntax.



```
pipeline {

}
```

- ### Node

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



```
node {

}
```

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

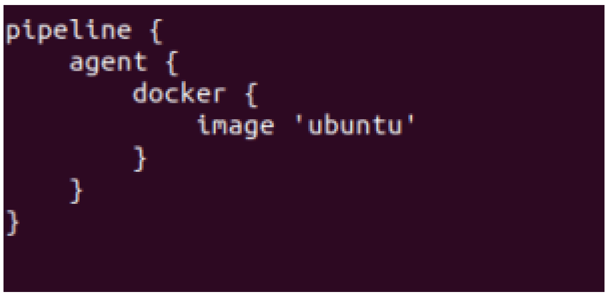
This parameter is applied at the root of the pipeline and it indicates that there is no global agent for the entire pipeline and each stage must specify its own agent.

- ### Label

Executes the pipeline/stage on the labelled agent.

- ### Docker

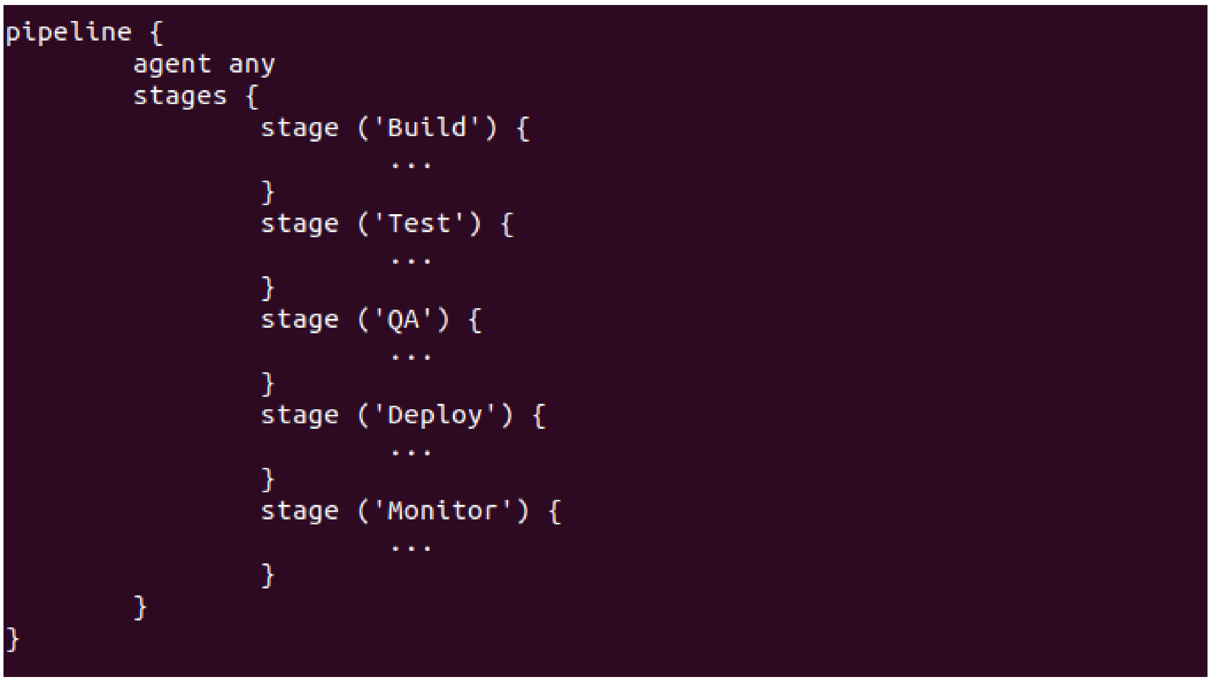
This parameter uses docker container as an execution environment for the pipeline or a specific stage. In the below example I'm using docker to pull an ubuntu image. This image can now be used as an execution environment to run multiple commands.



```
pipeline {
    agent {
        docker {
            image 'ubuntu'
        }
    }
}
```

- ## Stages

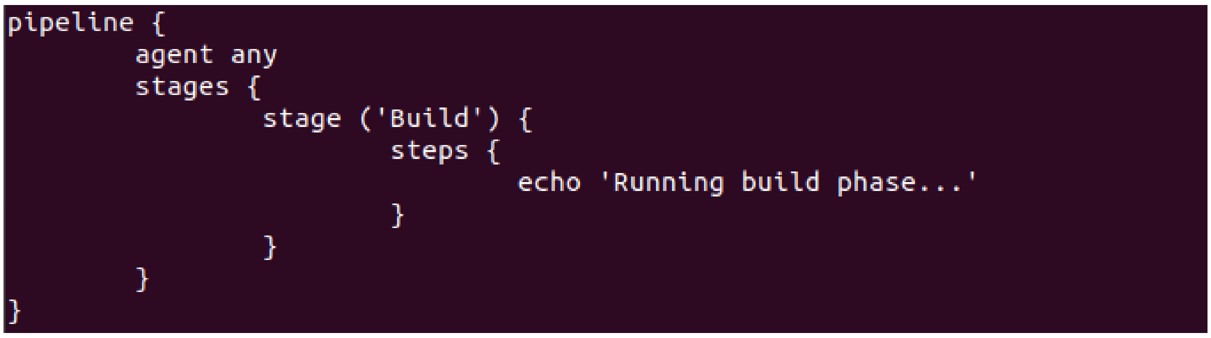
This block contains all the work that needs to be carried out. The work is specified in the form of stages. There can be more than one stage within this directive. Each stage performs a specific task. In the following example, I've created multiple stages, each performing a specific task.



```
pipeline {
        agent any
        stages {
                stage ('Build') {
                        ...
                }
                stage ('Test') {
                        ...
                }
                stage ('QA') {
                        ...
                }
                stage ('Deploy') {
                        ...
                }
                stage ('Monitor') {
                        ...
                }
        }
}
```

- **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.

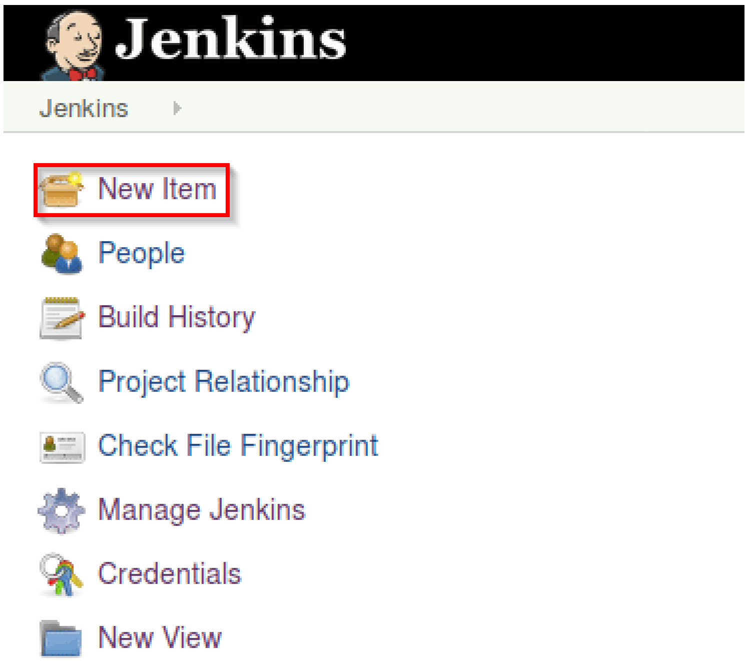


```
pipeline {
        agent any
        stages {
                stage ('Build') {
                        steps {
                                echo 'Running build phase...'
                        }
                }
        }
}
```

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.

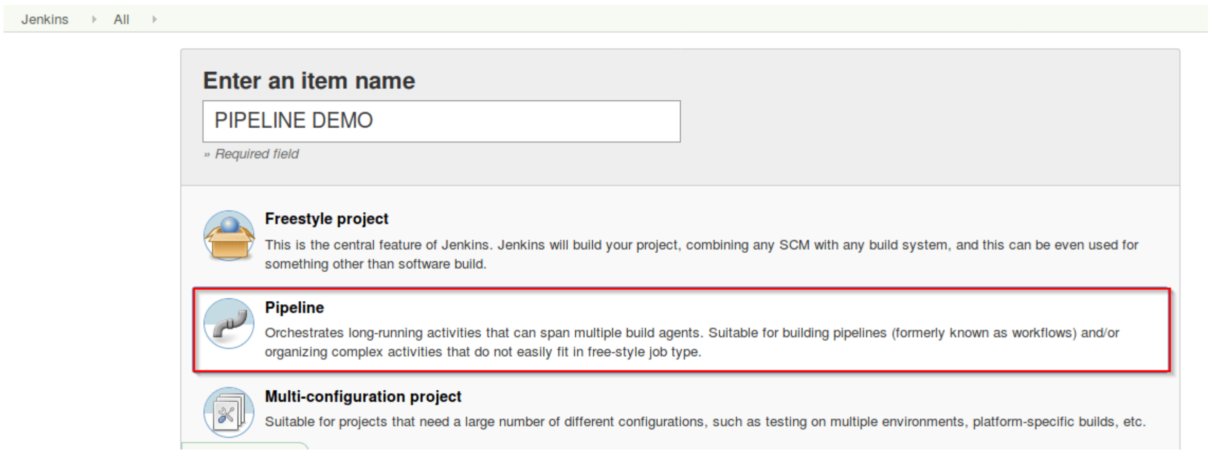
Creating your first Jenkins pipeline.

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 an item name

PIPELINE DEMO

» Required field

Freestyle project

This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

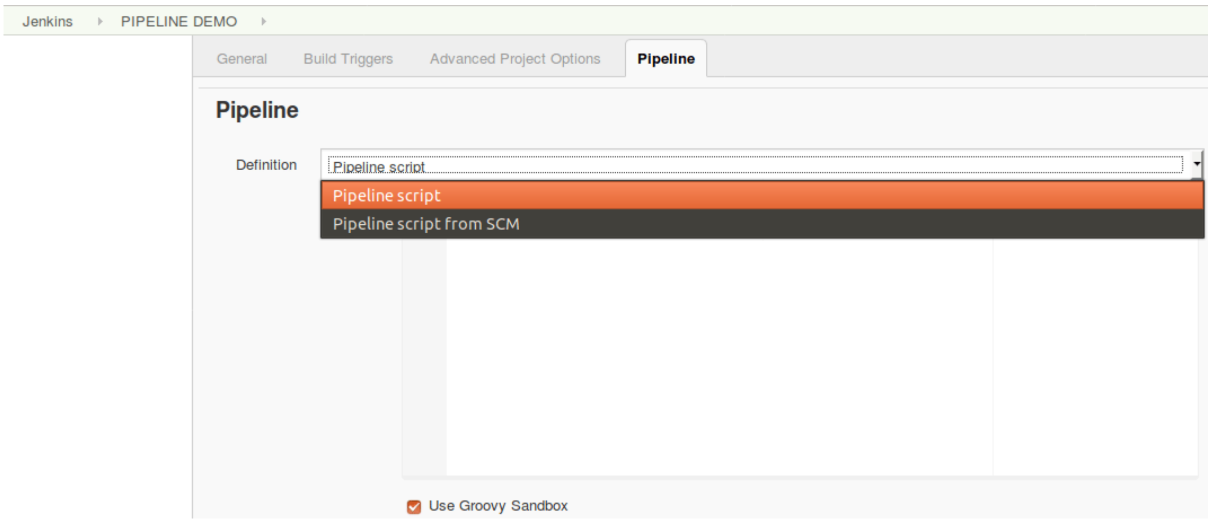
Pipeline

Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

Multi-configuration project

Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

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


Jenkins › PIPELINE DEMO ›

General | Build Triggers | Advanced Project Options | Pipeline

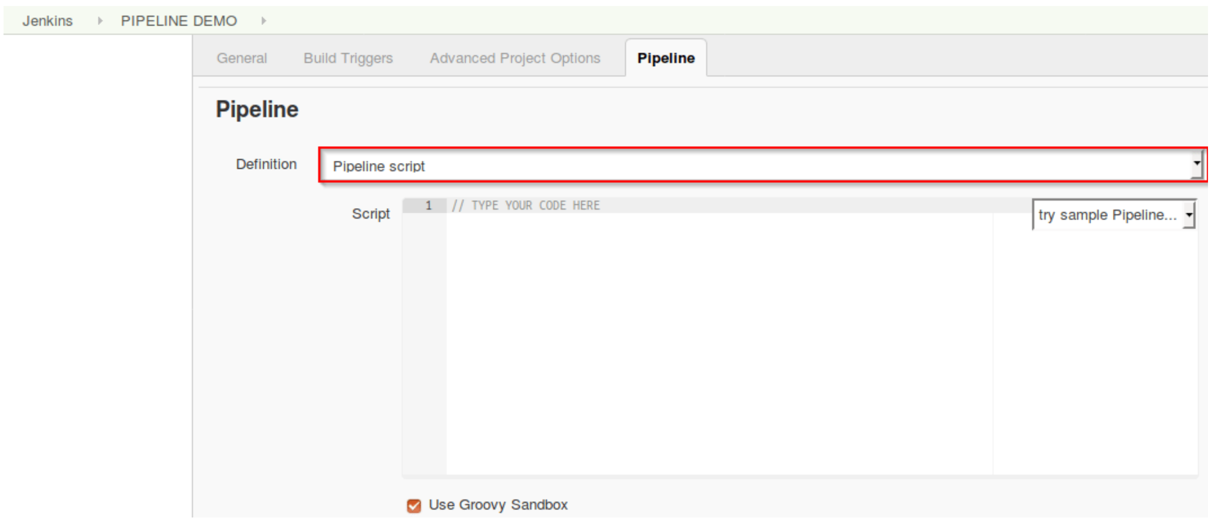
Pipeline

Definition: Pipeline script
- Pipeline script
- Pipeline script from SCM

☑ Use Groovy Sandbox

Declarative or scripted pipeline – Jenkins Pipeline Tutorial

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


Jenkins › PIPELINE DEMO ›

General | Build Triggers | Advanced Project Options | Pipeline

Pipeline

Definition: Pipeline script

Script:

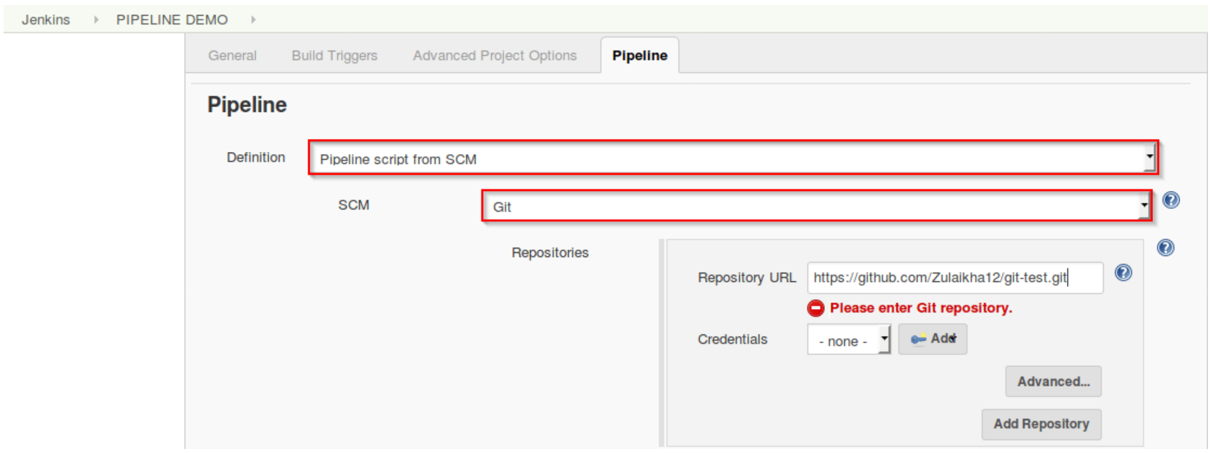
```
// TYPE YOUR CODE HERE
```

try sample Pipeline...

☑ Use Groovy Sandbox

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.


Jenkins › PIPELINE DEMO ›

Pipeline

Definition: Pipeline script from SCM

SCM: Git

Repositories

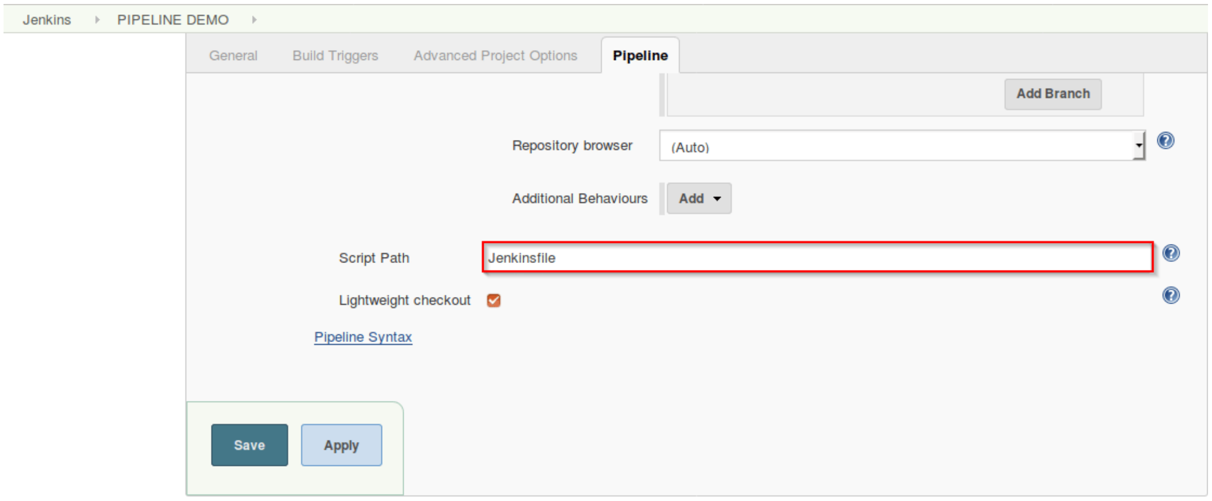
Repository URL: https://github.com/Zulaikha12/git-test.git

Please enter Git repository.

Credentials: - none -

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.


Jenkins › PIPELINE DEMO ›

Repository browser: (Auto)

Additional Behaviours: Add

Script Path: Jenkinsfile

Lightweight checkout: ✓

Script path – Jenkins Pipeline Tutorial

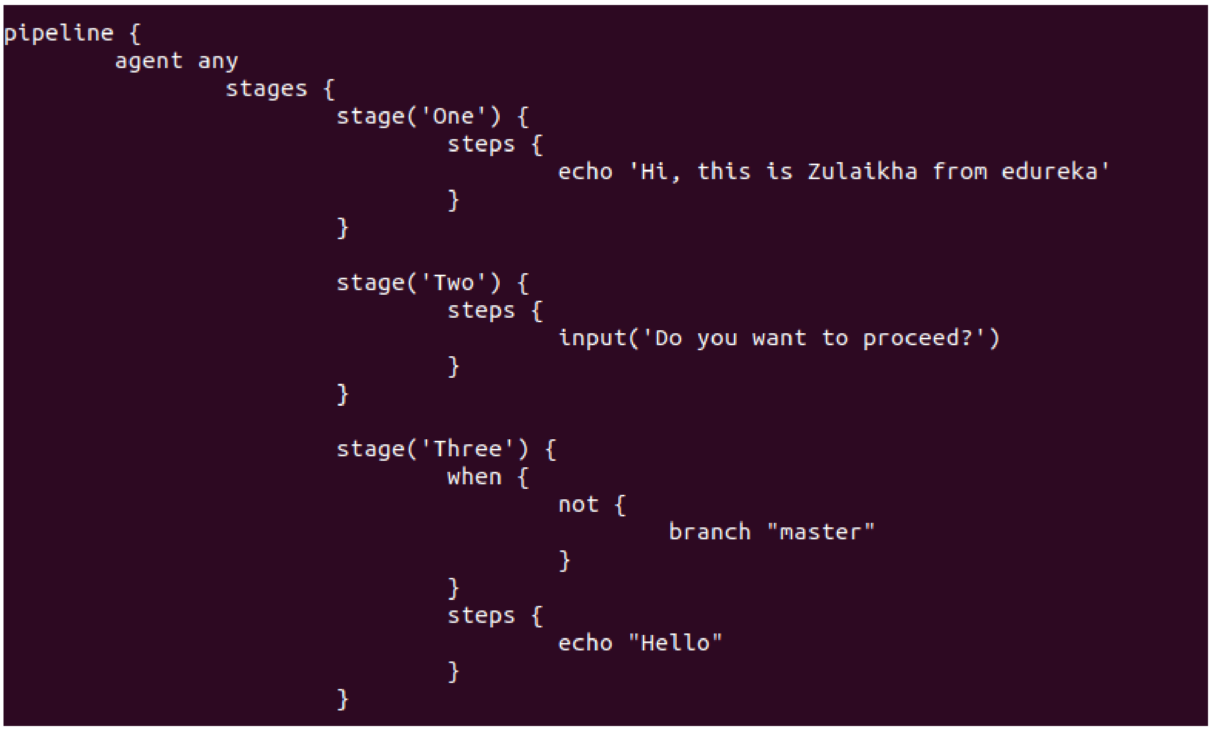
Now that you know how to create a pipeline, lets get started with the demo.

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.

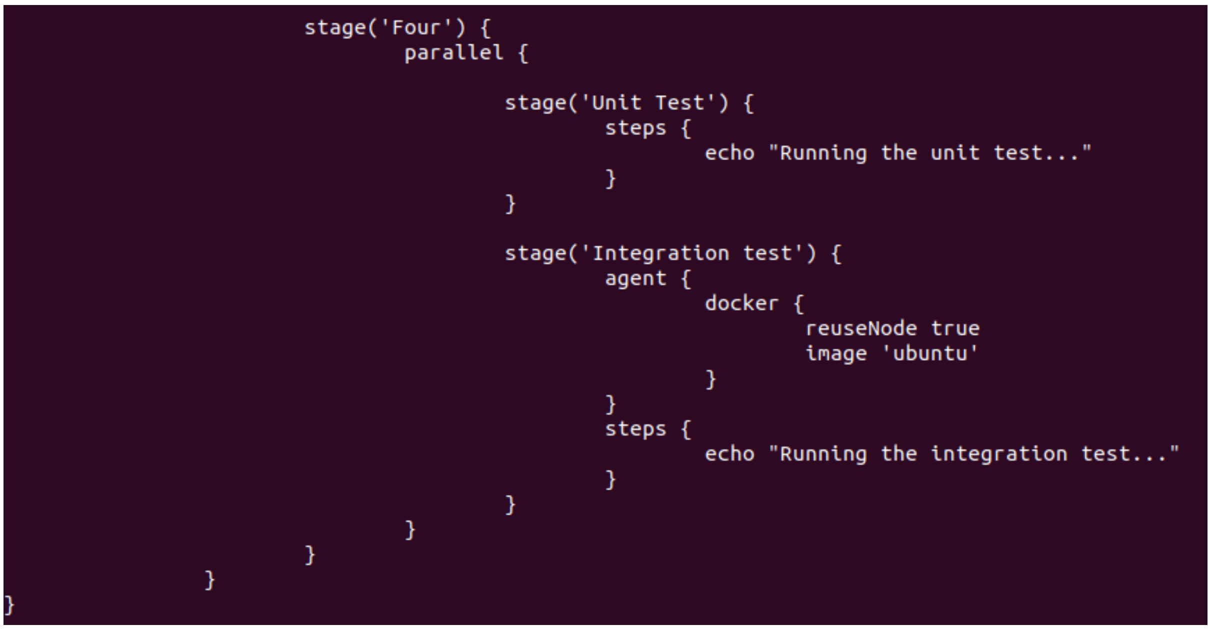
Since this is a declarative pipeline, I'm writing the code locally in a file named 'Jenkinsfile' and then pushing this file into my global git repository. While executing the 'Declarative pipeline' demo, this file will be accessed from my git repository. The following is a simple demonstration of building 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.



```
pipeline {
        agent any
                stages {
                        stage('One') {
                                steps {
                                        echo 'Hi, this is Zulaikha from edureka'
                                }
                        }

                        stage('Two') {
                                steps {
                                        input('Do you want to proceed?')
                                }
                        }

                        stage('Three') {
                                when {
                                        not {
                                                branch "master"
                                        }
                                }
                                steps {
                                        echo "Hello"
                                }
                        }
```

- 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.



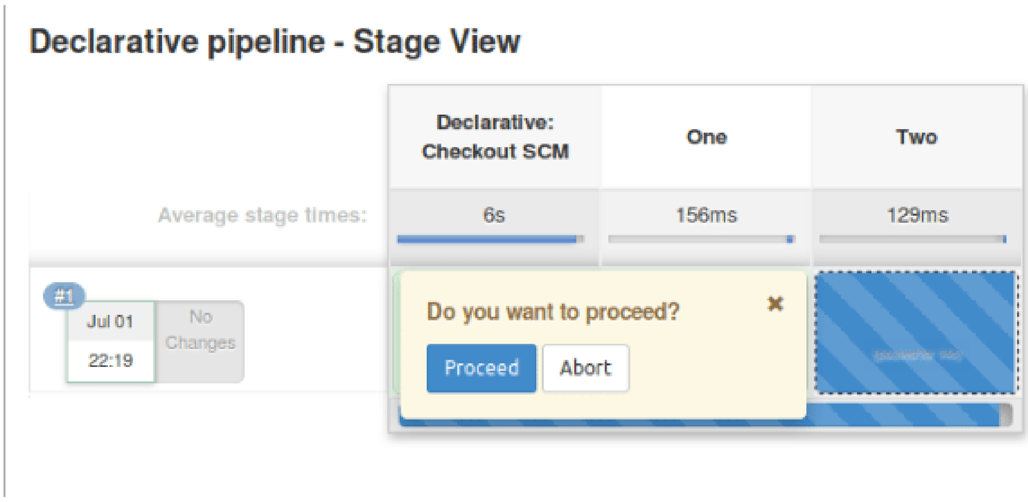
```
                        stage('Four') {
                                parallel {

                                        stage('Unit Test') {
                                                steps {
                                                        echo "Running the unit test..."
                                                }
                                        }

                                        stage('Integration test') {
                                                agent {
                                                        docker {
                                                                reuseNode true
                                                                image 'ubuntu'
                                                        }
                                                }
                                                steps {
                                                        echo "Running the integration test..."
                                                }
                                        }
                                }
                        }
                }
}
```

```
pipeline {
     agent any
     stages {
      stage('One') {
          steps {
echo 'Hi, this is Zulaikha from edureka'
           }
           }
      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..."
```

```
                                                          }
                                                        }
                                                        }
                                                        }
                                              }
                                    }
```

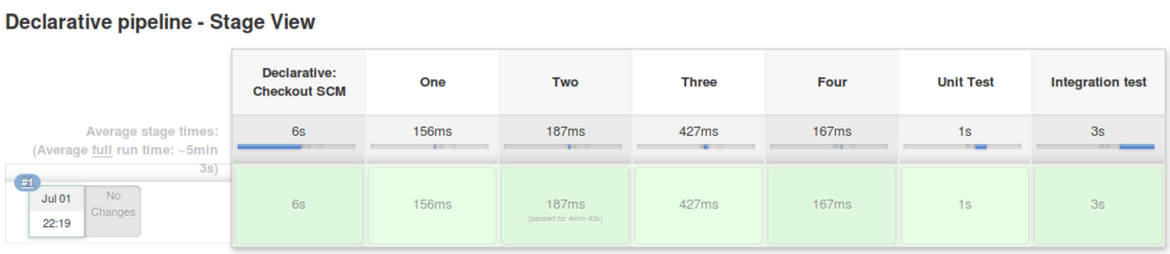
- Stage four runs a parallel directive. This directive allows you to run nested stages in parallel. Here, I'm running two nested stages in parallel, namely, 'Unit test' and 'Integration test'. Within the integration test stage, I'm defining a stage specific docker agent. This docker agent will execute 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.



Declarative pipeline - Stage View

	Declarative: Checkout SCM	One	Two
Average stage times:	6s	156ms	129ms
#1 Jul 01 22:19 No Changes			

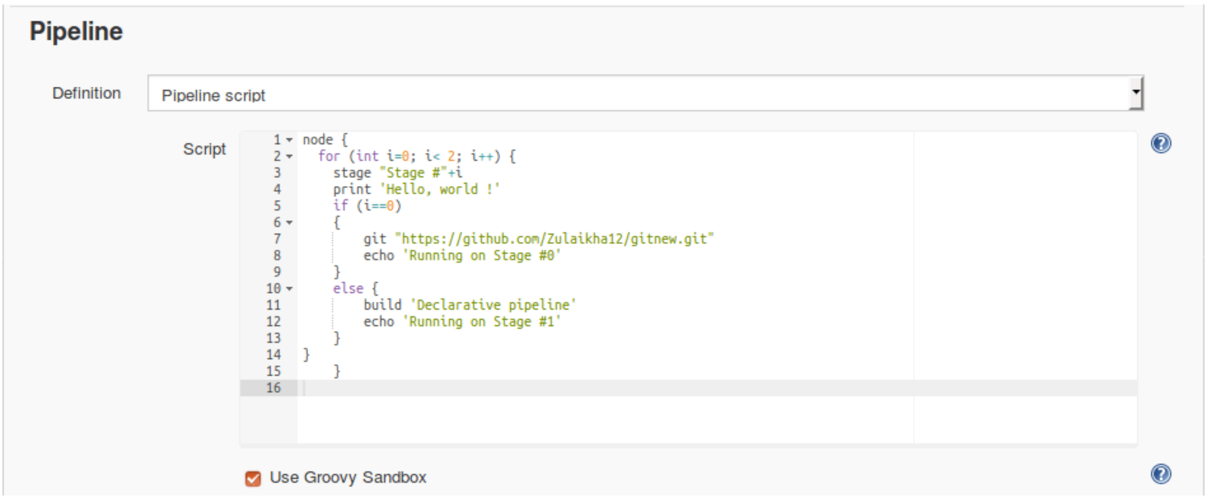


Declarative pipeline - Stage View

	Declarative: Checkout SCM	One	Two	Three	Four	Unit Test	Integration test
Average stage times: (Average full run time: ~5min 3s)	6s	156ms	187ms	427ms	167ms	1s	3s
#1 Jul 01 22:19 No Changes	6s	156ms	187ms (paused for 4min 43s)	427ms	167ms	1s	3s

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

Definition: Pipeline script

Script

```
node {
  for (int 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'
    }
}
    }
```

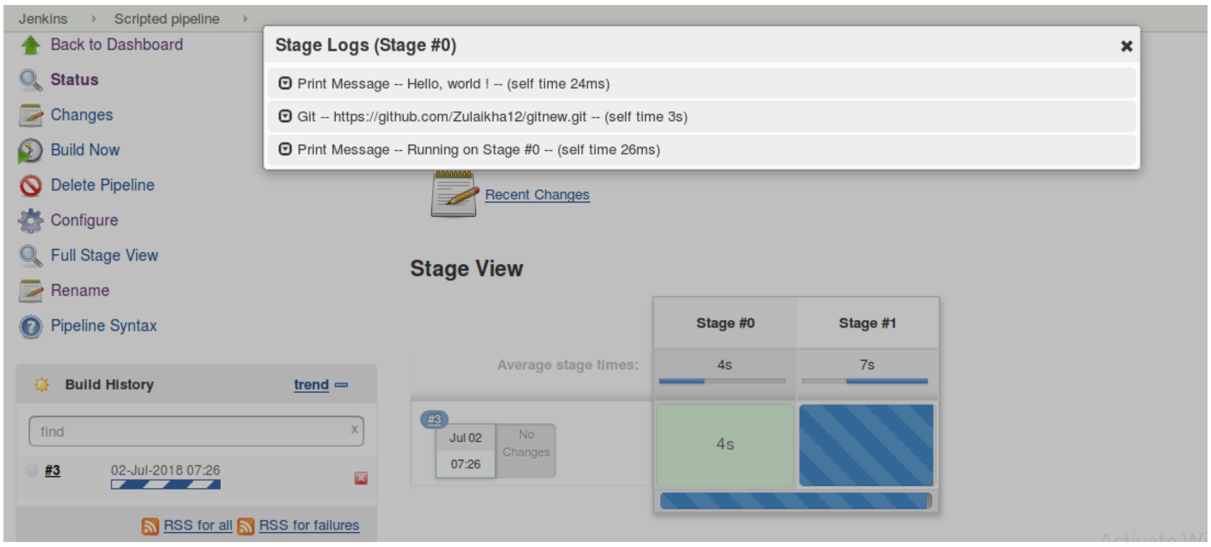
[✓] Use Groovy Sandbox

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.

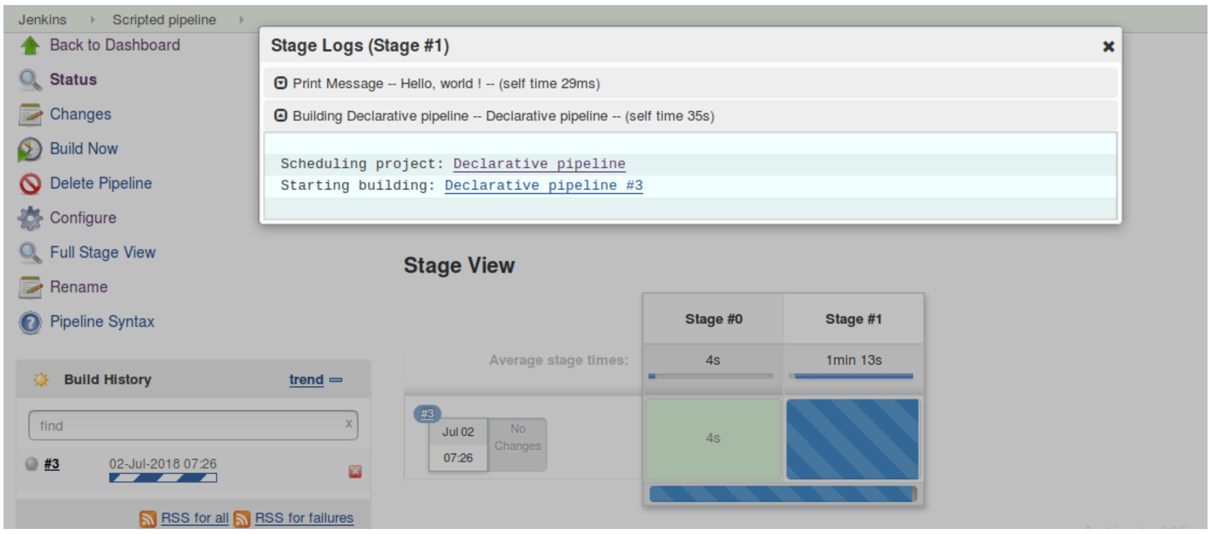
1. Shows the results of Stage #0



Stage Logs (Stage #0)

```
Print Message -- Hello, world ! -- (self time 24ms)
Git -- https://github.com/Zulaikha12/gitnew.git -- (self time 3s)
Print Message -- Running on Stage #0 -- (self time 26ms)
```

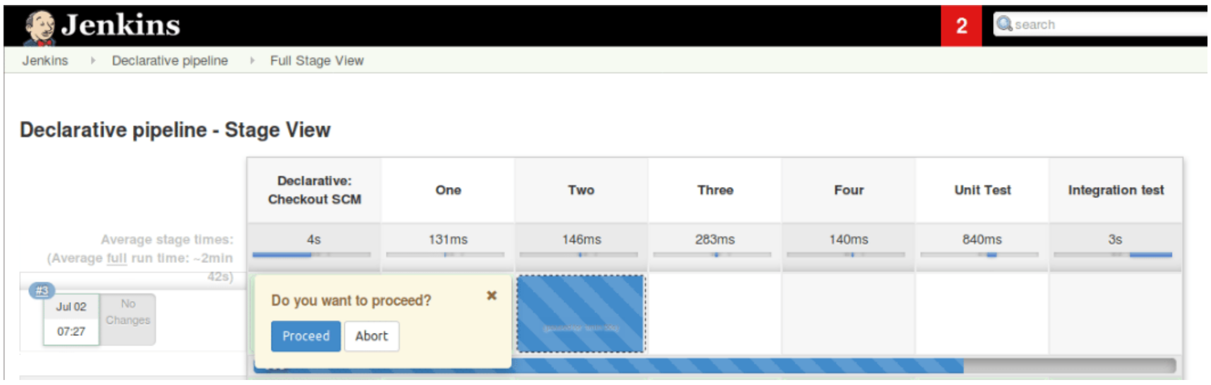
2. Shows the logs of Stage #1 and starts building the ‘Declarative pipeline’



Stage Logs (Stage #1)

```
Print Message -- Hello, world ! -- (self time 29ms)
Building Declarative pipeline -- Declarative pipeline -- (self time 35s)

Scheduling project: Declarative pipeline
Starting building: Declarative pipeline #3
```

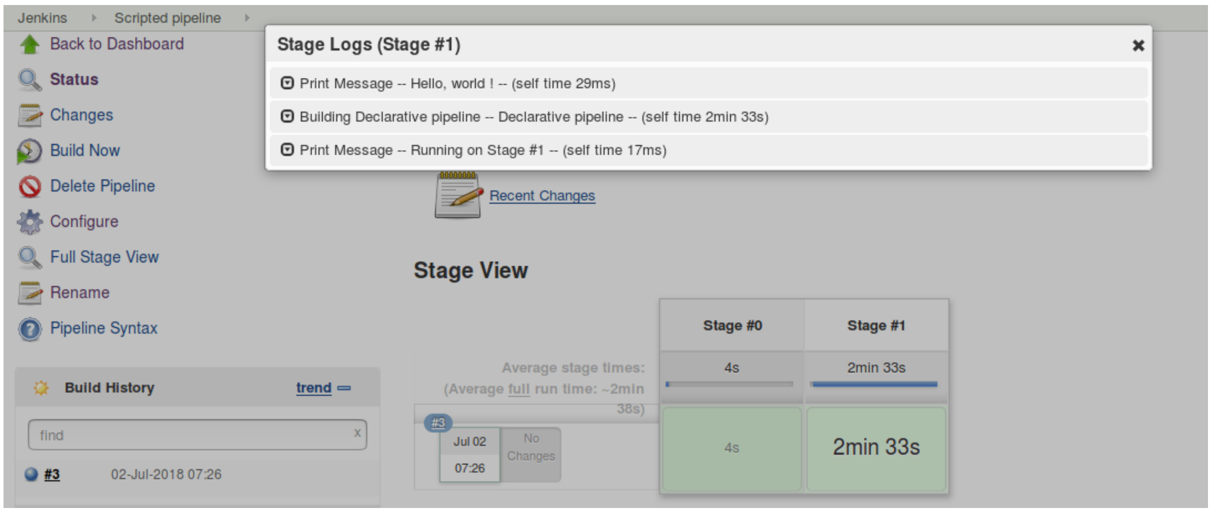
3. Execution of the ‘Declarative pipeline’ job.



Declarative pipeline - Stage View

	Declarative: Checkout SCM	One	Two	Three	Four	Unit Test	Integration test
Average stage times: (Average full run time: ~2min 42s)	4s	131ms	146ms	283ms	140ms	840ms	3s

4. Results.


Stage Logs (Stage #1)

Print Message -- Hello, world ! -- (self time 29ms)
Building Declarative pipeline -- Declarative pipeline -- (self time 2min 33s)
Print Message -- Running on Stage #1 -- (self time 17ms)