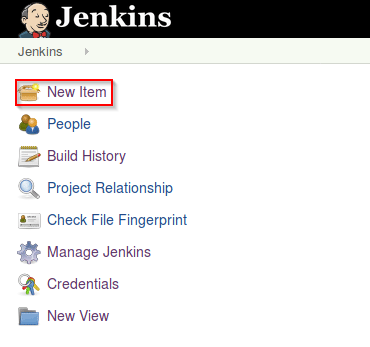
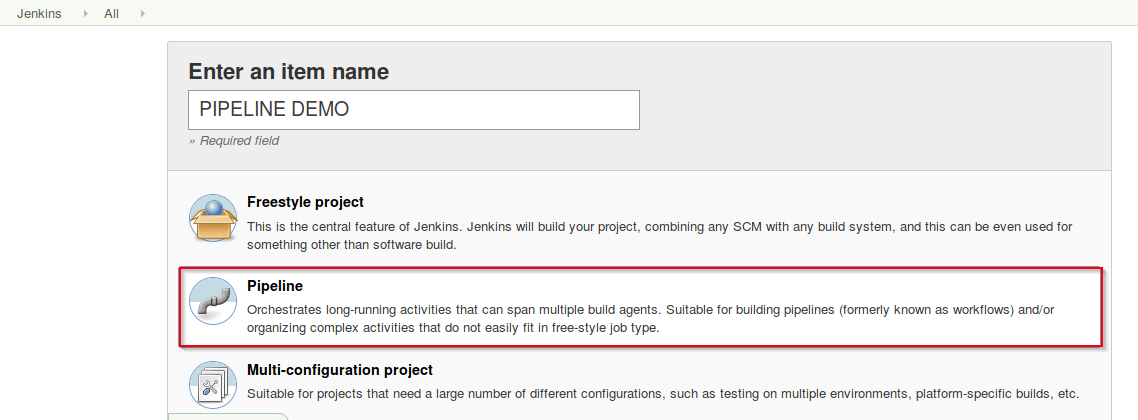
**Creating your first Jenkins pipeline.**

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

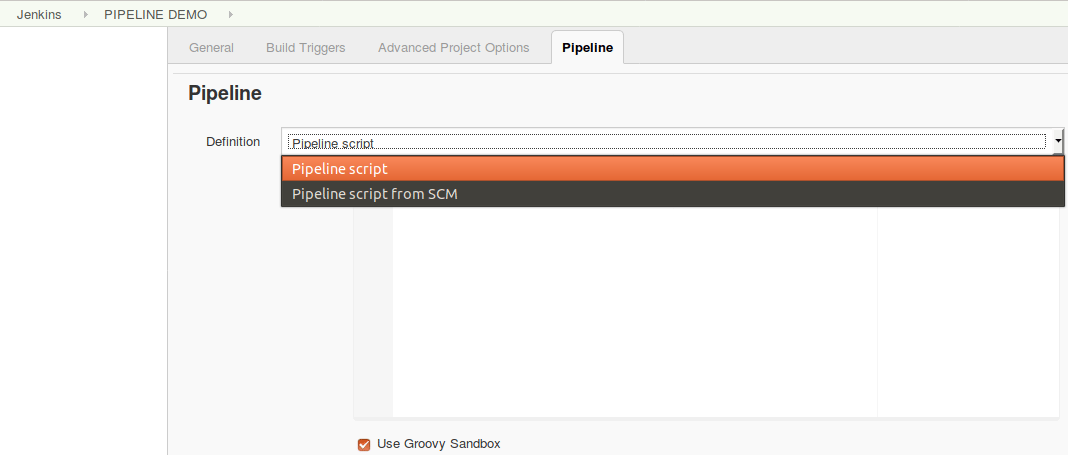


*Jenkins Dashboard – Jenkins Pipeline Tutorial*

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

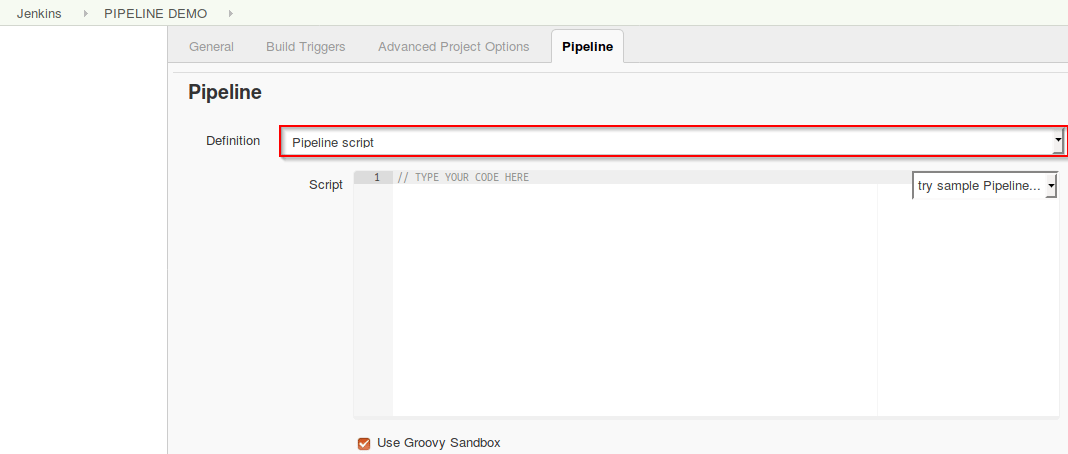
*Enter the project name – Jenkins Pipeline Tutorial*

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



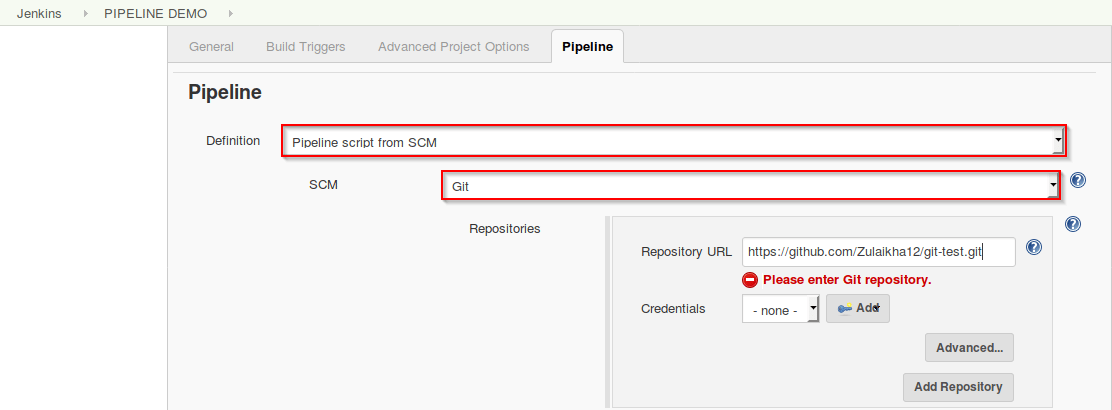
*Declarative or scripted pipeline – Jenkins Pipeline Tutorial*

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

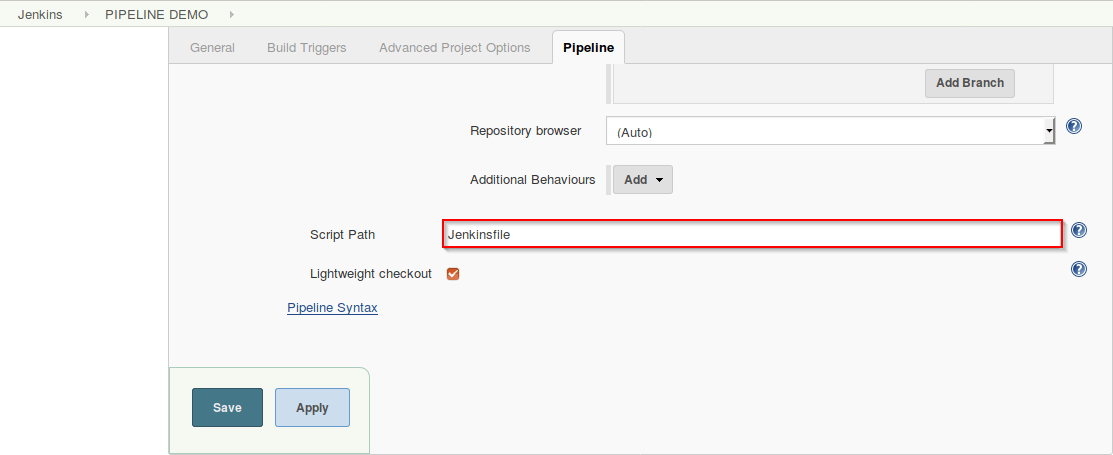


*Scripted Pipeline – Jenkins Pipeline Tutorial*

**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*

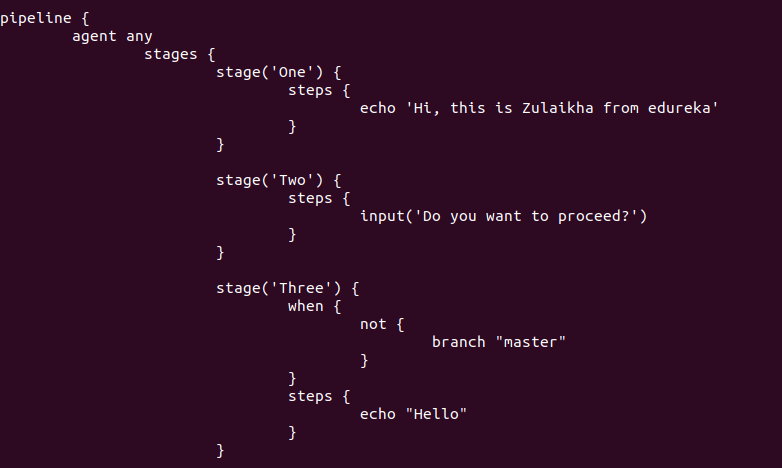
Now that you know how to create a pipeline, lets get started with the demo.

**Declarative Pipeline Demo**

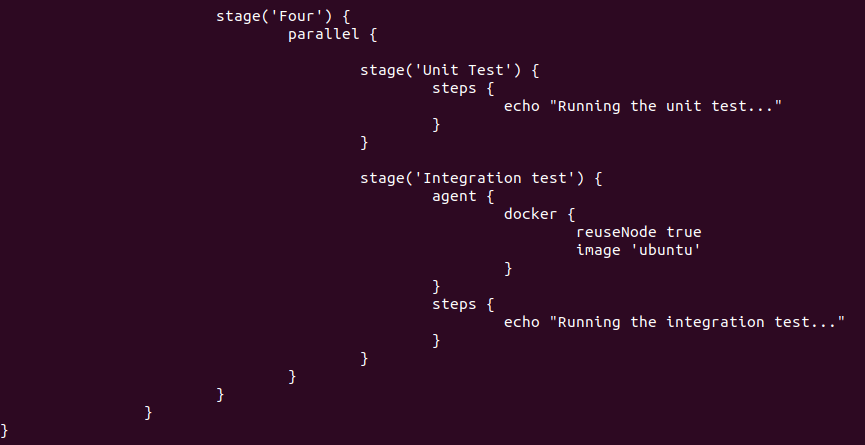
The first part of the demo shows the working of a declarative pipeline. Refer the above ‘Creating your first Jenkins pipeline’ to start. Let me start the demo by explaining the code I’ve written in my Jenkinsfile.

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.



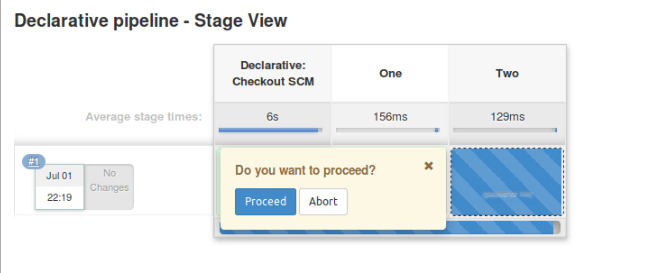
* Stage three runs a ‘when’ directive with a ‘not’ tag. This directive allows you to execute a step depending on the **conditions defined** within the ‘when’ loop. If the conditions are met, the corresponding stage will be executed. It must be defined at a stage level.
* In this demo, I’m using a ‘not’ 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.

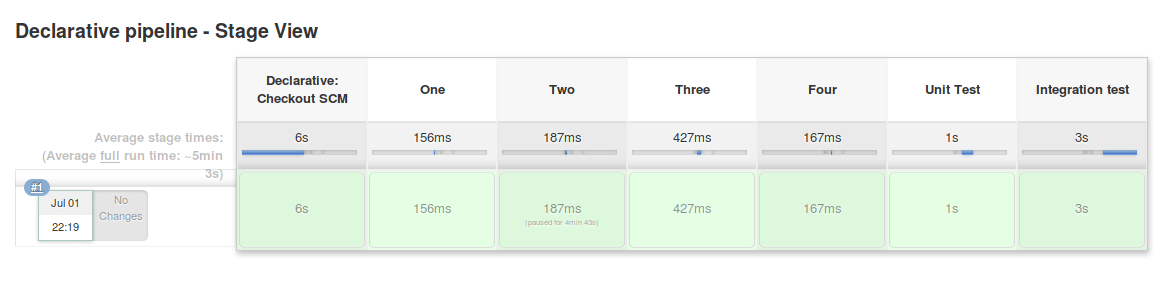


|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | pipeline {           agent any           stages {                   stage('One') {                   steps {                       echo 'Hi, this is Zulaikha from 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

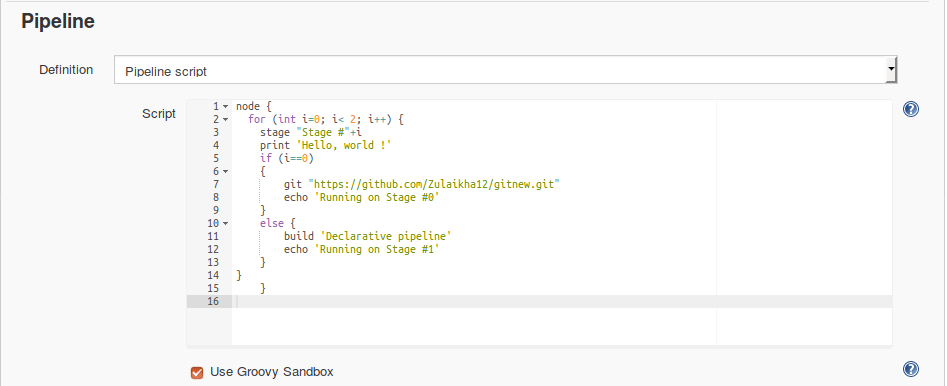
Now that I’ve explained the code, lets run the pipeline. The following screenshot is the result of the pipeline. In the below image, the pipeline waits for the user input and on clicking ‘proceed’, the execution resumes.

*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](https://www.edureka.co/blog/jenkins-pipeline-tutorial-continuous-delivery#Createpipeline) to create the scripted pipeline. I will run the following script.



Pipeline script – Jenkins Pipeline Tutorial

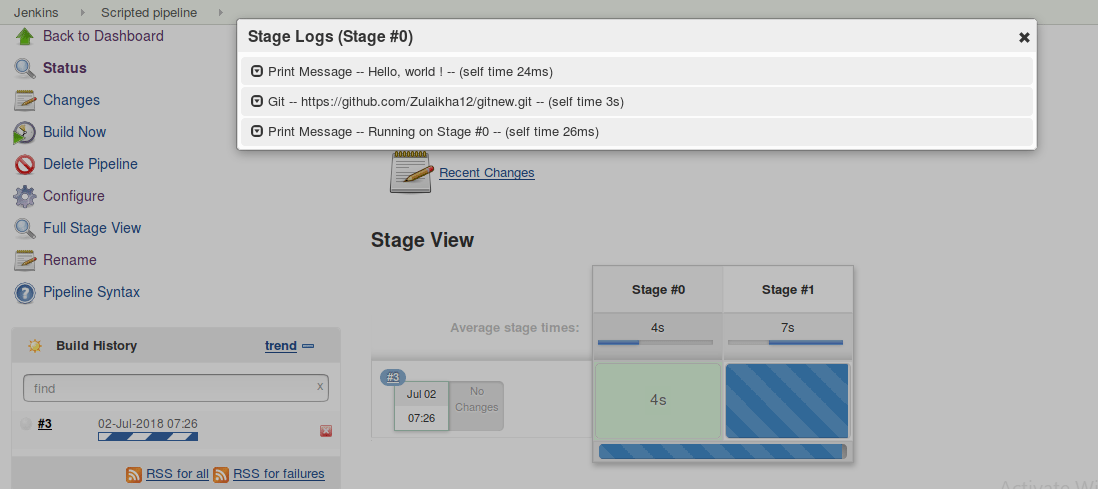
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | node {        for (i=0; i<2; i++) {             stage "Stage #"+i             print 'Hello, world !'             if (i==0)             {                 git "<a href="https://github.com/Zulaikha12/gitnew.git">https://github.com/Zulaikha12/gitnew.git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

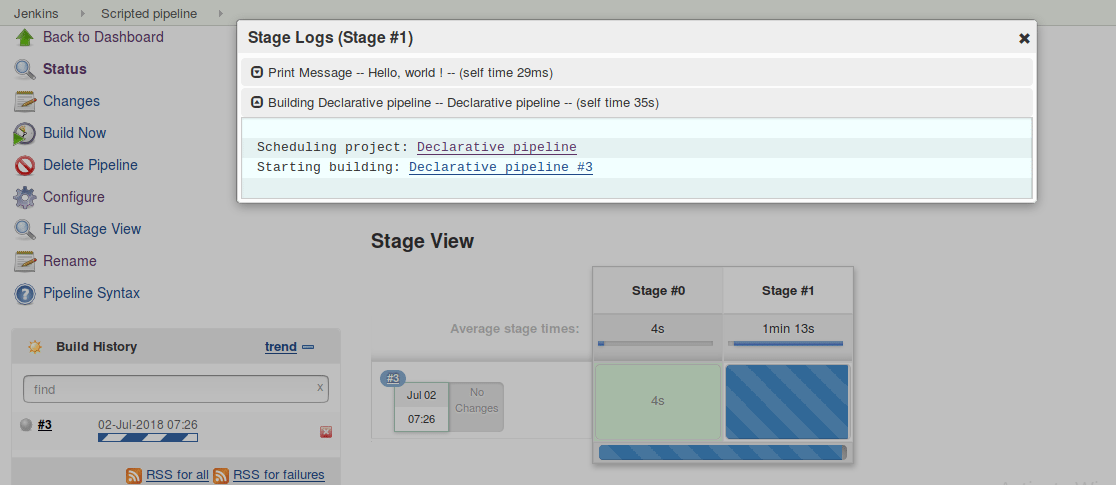
* The conditional ‘for’ loop. This for loop is for creating 2 stages namely, Stage #0 and Stage #1. Once the stages are created they print the ‘hello world!’ message
* Next, I’m defining a simple ‘if else’ statement. If the value of ‘i’ equals to zero, then stage #0 will execute the following commands (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

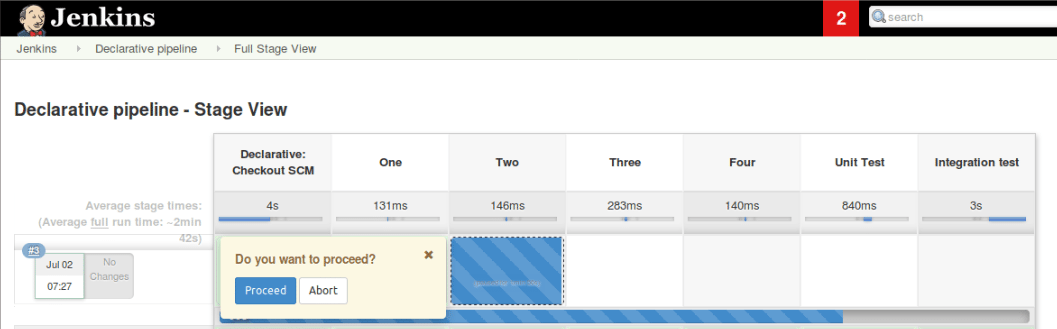
*Stage #0 – Jenkins Pipeline Tutorial*

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

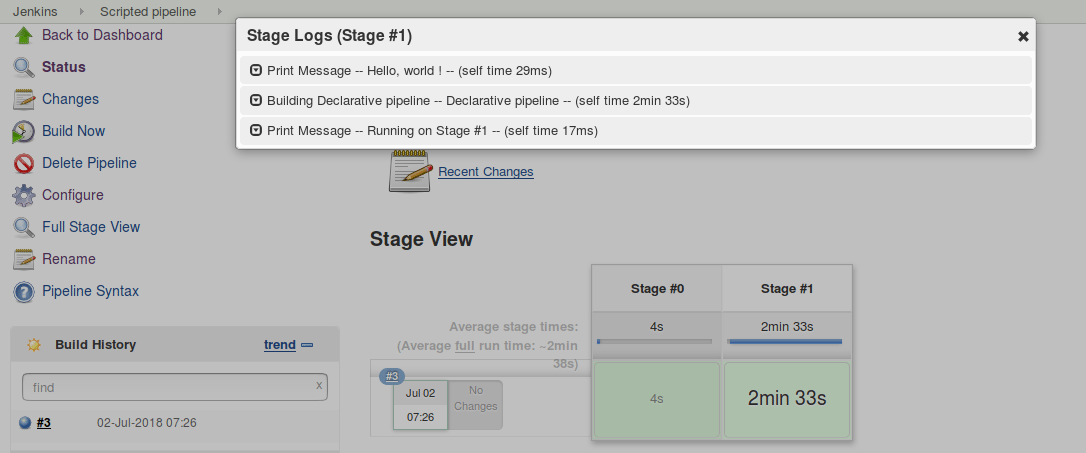


*Stage #1 running the declarative pipeline – Jenkins Pipeline Tutorial*

3. Execution of the ‘Declarative pipeline’ job.

*Execution of declarative pipeline-Jenkins Pipeline Tutorial*

4. Results.

*Final output-Jenkins Pipeline Tutorial*